TESTING AND PROFILING OF REGULAR TYPE OPERATIONS

DANIEL LOSCOS BARROSO

MÁSTER INTERUNIVERSITARIO EN MÉTODOS FORMALES EN INGENIERÍA INFORMÁTICA,
FACULTAD DE INFORMÁTICA, UNIVERSIDAD COMPLUTENSE DE MADRID
ESCUELA POLITÉCNICA SUPERIOR, UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INFORMÁTICOS,
UNIVERSIDAD POLITÉCNICA DE MADRID

Trabajo Fin Máster en Métodos Formales en Ingeniería Informática

Curso 2019-2020

Director y Colaboradores:

Manuel Hermenegildo Salinas
Pedro López García
José Francisco Morales Caballero

Convocatoria: Junio-Julio 2020
Calificación Obtenida: **SOBRESALIENTE 10**.

Autorización de difusión

Daniel Loscos Barroso

2-Julio-2020

El abajo firmante, matriculado en el Máster Interuniversiatio en Métodos Formales en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) y a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "TESTING AND PROFILING OF REGULAR TYPE OPERATIONS", realizado durante el curso académico 2019-2020 bajo la dirección de Manuel Hermenegildo Salinas, Pedro López García y José Francisco Morales Caballero en el Instituto IMDEA Software, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

Abstract

We performed an audit of the operations of the *regular types* library included with the Ciao pre-processor, CiaoPP, with the objective of exploring its correctness and efficiency. We centered our investigation on the operations relevant for performing type inference analysis via abstract interpretation, with special attention to the widening operators. We implemented tools to perform our white-box testing of the library, found the bottlenecks for analysis, and proposed some solutions to the main issues diagnosed in the investigation.

Keywords

Regular Types, Shape Analysis, Type Analysis, Abstract Interpretation, Widening Operators, Logic Programming, White-Box Testing.

Resumen en castellano

Hemos realizado una auditoria en la librería de *tipos regulares* del preprocesador de Ciao, CiaoPP, con el objetivo de explorar su corrección y eficiencia. Centramos nuestra investigación en las operaciones relevantes para el análisis de tipos por interpretación abstracta, prestando especial atención a los operadores de widening. Hemos implementado herramientas para realizar pruebas de caja blanca en la librería, encontrado los cuellos de botella para el análisis y hemos propuesto soluciones a los principales problemas detectados en la investigación.

Palabras clave

Tipos Regulares, Análisis de Tipos, Interpretación Abstracta, Operadores de Widening, Programación Lógica, Testing de Caja Blanca.

Contents

Acknkowledgements

I want to thank the IMDEA Software Institute and specially the Ciao Development team for allowing me to work with them and introducing me to their unique programming language.

I also want to acknowledge the work of all the people involved in the creation and implementation of this Master's degree for rightfully believing that this was a project worth investing on.

Finally I want thank the people in the intersection of both projects. Thank you for making this possible.

Chapter 1

Introduction

CiaoPP [12], the pre-processor of the Ciao [10] programming language, offers a large number of program analyses, based on abstract interpretation. Each such analysis is generally implemented as an "abstract domain," i.e., a plug-in to the general analysis framework. Among the abstract domains available there are a number devoted to the inference of the data structure shapes constructed by programs, and in particular to the inference of *regular types*, as finite representations of such data structures.

In this thesis we have performed an assessment of some of the *regular type* domains present in CiaoPP, and in particular on the library of regular type operations, `typeslib`. The focus has been to check its correctness and efficiency, with special attention to the different widening operators [3].

To achieve these goals, we have implemented tools for performing white-box testing of this regular type library.[1]

We also report on the problems and bottlenecks found, as well as the solutions that we propose for them.

We start by introducing the main concepts needed to place this work in context.

[1]The newest Ciao release is available at `https://ciao-lang.org` and the tools developed for this investigation are compiled in `https://github.com/LoscosBarroso/MTCode`.

1.1 Type Analysis in Logic Programming

Type theory is one of the main areas of Computer Science. Type systems help guarantee the correctness of software products, prevent undesired access to variables and help programmers clarify and understand code by giving variables and functions a clear and solid structure.

We can distinguish three groups of programming languages according to their type systems: there are untyped programming languages, such as pure $\lambda-$calculus, which don't even consider the notion of types; typed languages, such as C++ or Java where every operation argument and variable requires a type to be assigned (at compile or execution time); and those which, like Prolog, have some base types for arithmetic or I/O operations, which are generally checked dynamically, but are untyped for general purposes.

Type checking in (strongly) typed languages is usually performed before or during compilation to check for type inconsistencies in function calls or memory access. This way a lot of errors can be debugged even before the code is compiled, thus avoiding dangerous unexpected executions. In dynamic languages, type checks are generally performed at run time, based on mechanisms for marking data structures such as *tagging*.

In the realm of logic programming, the main representative, Prolog, is, as stated before, an untyped language, except for some base types, that are checked dynamically. There have also been some proposals for *strongly typed* logic programming languages, such as *Gödel* [13] or Mercury [31], which are more in the line of languages such as Haskell. Finally, the Ciao language takes an intermediate approach whose objective is to bridge the gap between the static and dynamic approaches, while preserving compatibility with traditional Prolog. This approach covers not just types but also many other properties that are useful for aiding program development and also program optimization, such as modes [23, 22], determinacy [18], nonfailure [1], types [28, 32], or cost [6, 5, 7, 29], as well as auto-documentation and testing. An assertion language [26] allows stating these properties of predicates or program points. Crucially, these assertions are voluntary, i.e., they are checked if they are present, but they are not required.

Such assertions are processed by the Ciao Preprocessor, CiaoPP [11, 25, 12], which is capable of finding non-trivial bugs statically or dynamically, verifying statically that programs comply with their specifications and performing many types of program optimizations.

The Ciao model can be considered an antecedent of the now-popular *gradual-* and *hybrid-typing* approaches [8, 30, 27].

Going back to the role of types in logic programming languages, it should be noted that, in pure logic programming languages there would in principle be no need for language-wide type systems since any atom can be given any desired typing just by adding structures of the form `type(atom)` or `type(functor(type1(arg1),...,typeN(argN)))`.

This, however, does not render such type systems useless in non-pure logic programming languages: types like *atom*, *int*, *number* being built in the language help write clearer code, avoid counter-intuitive contraptions, have efficient implementations for specific operations and detect failures earlier.

Once that has been established, the most interesting type-related analysis that one can perform on untyped logic programs is to infer the shapes of the data structures that the program generates during its execution. These data structures are built in programs by combining atoms and functors (including lists), and the basic types of the language, and are either finite or they typically have a regular, repeating structure that can be characterized finitely. The objective of analysis is to discover these regular structures, since they can provide very valuable information during program development [24, 11], for multiple uses such as detecting bugs, documentation, optimization, etc. This is what we call shape or inferred types analysis.

An intuitive idea would be that in typed programming languages type analysis checks that the types of the arguments of a function are the intended ones from the declaration (i.e., the expected types and signatures are defined and declared ahead of time by the programmer and the compiler just checks them); while in untyped programs the job is to infer all the possible types (data structures) that the program builds and that will be seen

at the arguments of predicates at call time.

This kind of type analysis provides an additional advantage over the ones generally established. It gives the programmer a notion of the actual data structures built by their code. That way, hidden patterns or structures that appear regularly as predicate arguments can be detected, optimized or made explicit by the developers.

The main difficulty derived from type analysis in logic programming languages comes from the synthesizing effort of abstracting data structures from a limited set of ground predicates. Eg.: When should we infer that a type is not "$4, 8, 15, 16, 23$ or 42" but *int*?

Specially in predicates with recursive clauses, if an analysis were to explore all possible reachable data structures without any synthesizing efforts, the list could very well be infinite; Eg.: a recursive predicate that decodes Peano numbers.

To achieve termination in such cases, analyzers generally employ *widening* [3] operations during type analysis. The choice of a widening operator is crucial for the performance and precision of an analysis, usually establishing a trade-off between those two desired properties [32].

1.2 Goals

The main goal of this thesis is to throughoutly study a library of regular type operations implemented to support type inference analysis via abstract interpretation; and obtain information on the correctness and efficiency of those operations.

We also aim to design and deploy tests that let us diagnose the different parts of the given implementation. This comes in two dimensions: the correctness of the operations and their efficiency for type inference analysis.

The secondary goal of this thesis is to improve the library under audit. After the study has been performed and some problems identified, we will consider possible solutions and their viability. Additionally, were they to be found during the investigation, we will try to fix any bugs or incorrectness in the operations.

1.3 Scope

This study will be performed in a white box environment over CiaoPP's [12] library for regular type operations: Typeslib. Many of the different abstract domains for type inference in Ciao [10] rely heavily on Typeslib's operations to do their job.

Therefore, we will limit our study to a specific system and implementation, for which we will have full access to the source code. This will enable the use of some invasive monitoring techniques as well as for precise time measurements of specific operations.

Due to the time constraints on the investigation, we made the diagnosis and profiling of problematic operations the key objective of our investigation. Some solutions or alternative implementations will be presented, proposed and explored; however they will not be tested to assess their viability. This is left as future work.

1.4 Main Contributions

The main contributions of this thesis are:

- We compiled an overview on regular types theory and representations.
- We provided an introduction to Ciao's regular types library and some of the type analysis domains that use it.
- We provided and overview on some widening operators for type definitions.
- We developed a property-based testing suite for regular type operations in Ciao.
- We developed two regular type generators to create test cases.
- We were able to identify undocumented preconditions and fix minor bugs in the management of the database of type operations.
- We audited the Ciao implementation of the *structural type widening* operator, fixing a bug that could result in non termination of type analysis.

- We developed a new Ciao package that allows for monitorization of targeted predicate calls at run time.
- We used this package to explore the efficiency of the different predicates involved in CiaoPP's *Eterms* regular type analysis.
- We could identify the main bottlenecks of the Eterms regular type analysis and came up with explanations for them.
- We came up with possible solutions to the identified problems: some to improve efficiency of certain type operations and others to achieve better efficiency/precision trade-offs in the analysis.
- We designed an efficient algorithm to identify if the union of a set of types belongs to the set.
- We designed a new widening operator that could potentially provide high-precision type analysis with a much better cost than Eterms. It combines bottom-up type analysis with defined types widening.

The rest of the thesis is structured as follows: in Section 2 we provide some theoretical background on regular types and their representation; in Section 3 we introduce CiaoPP's type analysis via abstract interpretation, its regular type operations library, and the different widening operators and domains it supports; in Section 4 we present the tools developed and experiments performed during our audit; improvement ideas and proposals are introduced in Section 5; finally, our conclusions of the investigation and possible lines of future work are compiled in Section 6.

Chapter 2

Theoretical Background

Before we start with the description of the study performed and its discussion there are some key concepts about regular type analysis that we would like to introduce to the reader. The main references for this section are the book "Tree Automata Techniques and Applications" [2] and the Master's Thesis [15] that gave birth to the Libvata library[1].

A ranked alphabet is a finite set of symbols $\mathcal{F}$ with an $Arity : \mathcal{F} \to \mathbb{N}$ function.[2] We define the **arity** of a symbol $f \in \mathcal{F}$ as $Arity(f)$. We also define $\mathcal{F}_k = \{f \in \mathcal{F} \mid Arity(f) = k\}$ and we assume the set of **constants** ($\mathcal{F}_0$) to be non-empty. We use the notation f/k to indicate that a symbol $f \in \mathcal{F}$ has arity k.

Variables are a set $\mathcal{X}$ of constants such that $\mathcal{F}_0 \cap \mathcal{X} = \emptyset$. Unlike the symbols of a ranked alphabet, variables do not have an arity.

The set of **terms** over a ranked alphabet $\mathcal{F}$[3] and a set of variables $\mathcal{X}$ is the smallest set $T(\mathcal{F}, \mathcal{X})$ defined recursively by:

- $\mathcal{F}_0 \in T(\mathcal{F}, \mathcal{X})$,
- $\mathcal{X} \in T(\mathcal{F}, \mathcal{X})$ and
- if $p \geq 1$, $f \in \mathcal{F}_p$ and $t_1, \ldots, t_p \in T(\mathcal{F}, \mathcal{X})$, then $f(t_1, \ldots, t_p) \in T(\mathcal{F}, \mathcal{X})$.

[1] A C++ library for efficient manipulation with non-deterministic finite (tree) automata. Its source code and installation guidelines can be found at https://github.com/ondrik/libvata.

[2] We assume that $0 \in \mathbb{N}$.

[3] We abuse the notation of $\mathcal{F}$ as a set of symbols to carry their arity function as well.

The set of **ground terms** over a ranked alphabet $\mathcal{F}$ is defined as $T(\mathcal{F}, \emptyset)$ and usually addressed as $T(\mathcal{F})$. We say that a specific term $t \in T(\mathcal{F}, \mathcal{X})$ is **linear** if each variable occurs at most once in t.

2.1 Regular Types

A **type** $\alpha \subset T(\mathcal{F})$ is a subset of the ground terms built from a ranked alphabet $\mathcal{F}$ and can be represented using different formalisms like tree automata [2, 15], type terms and term grammars [4], predicates [33], or graphs [14].

The ground terms that conform a type can also be seen as the language of their representation (the set of every ground term instance of $T(\mathcal{F})$ that adheres to that representation) in any given formalism, this is what we call a **type definition**.

We can therefore define **regular types** (types which have very desirable analytic properties) using any of the formalisms used for type representation. For example, when represented by their defining tree automata: a type is a regular type if its components constitute a regular tree language; that is, the language recognized by a non-deterministic finite tree automaton.

In the following sections 2.2 and 2.3 we will take a closer look at two of these formalisms, to help the reader understand these and other equivalent definitions of "regular type".

2.2 Regular Tree Languages

It should now be clear to the reader why we can refer to terms as **trees**: since the arity of every symbol $f \in \mathcal{F}$ is finite, one can see terms as finite ordered ranked trees, where the leaves are labeled with variables or elements of $\mathcal{F}_0$ and the rest of the nodes are represented by symbols whose arity is equal to the number of children of their node.

In the same way as a regular language is the set of symbol strings accepted by a finite automaton, a regular tree language is the set of ground terms accepted by a finite tree automaton according to the following definitions:

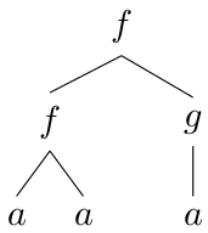

Figure 2.1: *Example ground term* $t = f(f(a,a),g(a))$ *over the ranked alphabet* $\mathcal{F} = \{a/0,\ g/1,\ f/2\}$ *viewed as a tree. Extracted from [15].*

A **tree language** is a set of trees $\mathcal{L} \in T(\mathcal{F})$ over the same ranked alphabet $\mathcal{F}$. We say that $\mathcal{L}$ is a **regular tree language** if a **Non-deterministic Finite Tree Automaton (NFTA)** $\mathcal{A}$ exists such that the set $\mathcal{L}(\mathcal{A})$ of all accepted ground terms by $\mathcal{A}$, is equal to $\mathcal{L}$.

A bottom-up NFTA $\mathcal{A}$ over $\mathcal{F}$ is defined by $\mathcal{A} = (\mathcal{Q}, \mathcal{F}, \mathcal{Q}_f, \Delta)$ where $\mathcal{Q}$ is a finite set of states (symbols of arity 1) disjoint with $\mathcal{F}$, $\mathcal{Q}_f \subseteq \mathcal{Q}$ is the subset of final states and Δ the set of valid transition rules of the form: $f(q_1(t_1),...,q_k(t_k)) \underset{\mathcal{A}}{\rightarrow} q(f(t_1,...,t_k))$ or $g \underset{\mathcal{A}}{\rightarrow} q(g)$, where $t_1,...,t_k \in \mathcal{X},\ f/k, g/0 \in \mathcal{F}$ and $q, q_1,...,q_k \in \mathcal{Q}$.

These clauses define the move relation $t \underset{\mathcal{A}}{\rightarrow} t'$ for the set $T(\mathcal{F} \cup \mathcal{Q})$. $t \underset{\mathcal{A}}{\rightarrow} t'$ is a valid move if:

- t is equal to t' at every node except for the sub-trees s and s', while those verify $s \underset{\mathcal{A}}{\rightarrow} s'$.
- $t = g/0 \in \mathcal{F}$, $t' = q(g)$, $q \in \mathcal{Q}$ and a rule $g \underset{\mathcal{A}}{\rightarrow} q(g) \ \in \Delta$ exists.
- or $t = f(q_1(u_1),...,q_k(u_k))$, $t' = q(f(u_1,...,u_k))$ where $f/k \in \mathcal{F}$, $u_1,...,u_k \in T(\mathcal{F})$, $q, q_1,...,q_k \in \mathcal{Q}$ and a rule $f(q_1(t_1),...,q_k(t_k)) \underset{\mathcal{A}}{\rightarrow} q(f(t_1,...,t_k)) \ \in \Delta$ exists.

We also define $\underset{\mathcal{A}}{\rightarrow}^*$ as the reflexive and transitive closure of $\underset{\mathcal{A}}{\rightarrow}$. A ground term $t \in T(\mathcal{F})$ is **accepted** by the automata $\mathcal{A}$ if and only if $t \underset{\mathcal{A}}{\rightarrow} q(t)$ for some state $q \in \mathcal{Q}_f$.

As we said previously, the set of all ground terms accepted by NFTA $\mathcal{A}$ constitutes the **language** $\mathcal{L}(\mathcal{A})$. Two tree automata are equivalent if they have the same language. As in the case of word automata, there are also bottom-up NFTA with $\epsilon-$transitions and Deterministic Tree Automata, with the same expressive power.

Finally, a top-down NFTA $\mathcal{A}$ over $\mathcal{F}$ is defined by $\mathcal{A} = (\mathcal{Q}, \mathcal{F}, \mathcal{I}, \Delta)$ where $\mathcal{Q}$ is a finite set of states (symbols of arity 1) disjoint with $\mathcal{F}$, $\mathcal{I} \subseteq \mathcal{Q}$ is the subset of initial states and Δ the set of valid transition rules of the form: $q(f(t_1, ..., t_k)) \underset{\mathcal{A}}{\rightarrow} f(q_1(t_1), ..., q_k(t_k))$ or $q(g) \underset{\mathcal{A}}{\rightarrow} g$, where $t_1, ..., t_k \in \mathcal{X}$, $f/k, g/0 \in \mathcal{F}$ and $q, q_1, ..., q_k \in \mathcal{Q}$. While the move relation is similar to the one for bottom-up NFTA, in this case, the acceptance of a tree t in the language is established if $\exists q \in \mathcal{I}$ such that $q(t) \underset{\mathcal{A}}{\rightarrow}^* t$.

2.3 Dart & Zobel Regular Types

In [4] Dart and Zobel define **regular types** as the types that can be specified by a regular term grammar. Regular types are closed under the union operation, and are partially ordered under the inclusion operator since some types are subsets of others. Dart and Zobel define them as not tuple-distributive and the regular type library of CiaoPP, Typeslib, is strongly rooted in this article [4] presented in the book [24].

We now present the relevant definitions for working with regular types:

First we assume the existence of a set of **function symbols** with their associated arity: $f/1$, $g/2, ...$, $a/0$, $b/0$, etc.; a set of **variables** x, y, z, etc. and an infinite set of **type symbols**: α, β, γ, δ... Function symbols of 0 arity are also called **constants**.

We define **type terms** inductively as a variable, a constant, a type symbol or as $f(\tau_1, ..., \tau_n)$ where f/n is a function symbol and $\tau_1, ..., \tau_n$ are type terms. A **pure type term** is a type term with no variables, and a **logical term** is a type term without type symbols.

We assume a partition of the set of constants that defines the **base types**, these types will be used to define all other types. This is how base types such as *integer* or *character* are defined. We also associate a type symbol α and a predicate definition to each base type so that $\alpha(c)$ is true if and only if c is a constant of the base type of α.

Other, non-basic types, are defined by **type rules** that associate types with predicate definitions. A type rule of the form $\alpha \rightarrow \Upsilon$ where α is a type symbol and Υ a set of pure

type terms; is associated with a set of clauses defining the predicate α which for each pure type term $\tau \in \Upsilon$ has a clause: $\alpha(t) \leftarrow \beta_1(x_1) \wedge ... \wedge \beta_k(x_k)$. Where t is τ after replacing each appearance of β_k by x_k.

A **regular term grammar** T is a set of type rules. An example given in [4] to illustrate this concept is the regular term grammar defined by the rules $\alpha \rightarrow \{a, b\}$ and $\beta \rightarrow \{\text{NIL}, tree(\beta, \alpha, \beta)\}$ where α and β are type symbols. Their associated predicate definitions would be: $\alpha(a) \leftarrow$, $\alpha(b) \leftarrow$, $\beta(\text{NIL}) \leftarrow$ and $\beta(tree(x_1, x_2, x_3)) \leftarrow \beta(x_1) \wedge \alpha(x_2) \wedge \beta(x_3)$

This way an equivalence is established between type rules and type definitions in type theory from Lloyd in [17].

A type symbol α is defined by a set T of type rules when there exists a rule $(\alpha \rightarrow \Upsilon) \in T$. A type term τ is defined by a set T of type rules when each type symbol in τ is defined in T. Let T be a set of type rules that defines the type symbols $\alpha_1, ..., \alpha_k$, we define Φ_T as the definite program that contains the predicates and clauses associated with the type rules defining $\alpha_1, ..., \alpha_k$ and M_{Φ_T} as the least model of said program.

We are now ready to define **regular types**. Informally, $[\tau]_T$ is the set of ground terms that can be derived form a pure type term τ by applying the rules of T. Formally, let T be a set of type rules; the type associated with the pure type term τ with respect to T given by the recursive definition:

$$[\tau]_T = \begin{cases} \{c\} & \text{if } \tau \text{ is a constant symbol } c. \\ \{t \mid \alpha(t) \in M_{\Phi_T}\} & \text{if } \tau \text{ is a type symbol } \alpha. \\ \{f(t_1, ..., t_n) \mid t_i \in [\tau_i]_T, \forall i = 1, ..., n\} & \text{if } \tau \text{ is a pure type term } f(\tau_1, ..., \tau_n). \end{cases}$$

is called a **regular type**.

2.4 Tuple-Distributive Regular Types

The notion of tuple-distributivity is due to Mishra [21]. **Tuple-distributive regular types** are regular types closed under tuple distributive closure [21, 33]. Intuitively, the tuple-distributive closure of a set of terms is the set of all terms constructed recursively by

permuting each argument position among all possible terms that have the same function symbol [33]. For example let the set $\mathcal{L} = \{pair(a, b), pair(c, d)\}$, then its tuple-distributive closure would be $\mathcal{L}' = \{pair(a, b), pair(a, d), pair(c, b), pair(c, d)\}$.

The class of tuple-distributive regular types is not closed under type union ($\mathcal{L} = \{pair(a, b), pair(c, d)\}$ is not tuple-distributive while $\mathcal{L}_1 = \{pair(a, b)\}$ and $\mathcal{L}_2 = \{pair(c, d)\}$ are), and is strictly less powerful than the class of regular types [33]. We now give a definition of tuple-distributive regular types based on their tree automaton representation:

A **tuple-distributive regular type (TDRT)** is a set of terms which is the language accepted by a bottom-up NFTA such that its set of rules does not contain two rules of the form: $f(q_1, \ldots, q_n) \to q$ and $f(q'_1, \ldots, q'_n) \to q$, i.e., there are not two rules with the same target state (right hand side), whose left hand sides have the same main function symbol.

This concept is of upmost importance because some of Typelib's operations assume the absence of tuple distributivity and provide wrong answers if this assumption is broken.

Another example is the operation given by Dart and Zobel for testing regular type inclusion in [4]. Lunjin Liu proved in [19] that their algorithm for inclusion checking was complete but incorrect for regular types in general. This algorithm is only proofed to be correct for non tuple-distributive regular types.

Since Typeslib implements most regular type operation in a similar fashion to the ones described by Dart and Zobel, and since the inclusion operation is basic to most of the others, Typeslib is only correct for non tuple-distributive regular types.

Chapter 3

Regular Type Inference Analysis Domains in CiaoPP

As mentioned before, CiaoPP uses abstract interpretation for most of its analyses and in particular for the different type inference analyses. One of the main analysis domains for inferring regular types is the Eterms domain [32]. The basic ideas for this analysis can be found in [32] and we will introduce them after these definitions:

We define $\gamma(T)$ as the set of all concretizations of T (the set of ground terms represented by that term grammar) and $\gamma : \mathcal{G} \rightarrow T(\mathcal{F})$ as the concretization function. We also define the equivalence relationship $\equiv$ as $T_1, T_2 \in \mathcal{G}$ verify $T_1 \equiv T_2 \iff \gamma(T_1) = \gamma(T_2)$.

In abstract interpretation type derivation analysis, a type (represented by its term grammar) is used as the abstract representation of a set of terms. Given a set of variables $\{x_1, ..., x_k\}$ (representing the arguments for predicates in the code, for example) any term substitution $\theta = \{x_1 \leftarrow t_1, ..., x_k \leftarrow t_k\}$ can be approximated by an **abstract substitution** $\omega = \{x_1 \leftarrow T_{x_1}, ..., x_k \leftarrow T_{x_k}\}$ where $\forall i \mid\mid t_i \in \gamma(T_{x_i})$. This can be represented as $\langle T_1, ..., T_k \rangle$ or T_n for short.

We also introduce the distinguished bottom abstract substitution $\bot$ that verifies $\gamma(\bot) =$ as the representation of any $\langle T_1, ..., T_k \rangle$ such that $\exists T_i = \bot$ and is of course the bottom-most element of the abstract substitution lattice. The top-most element $\top$ is defined as $\langle T_1, ..., T_k \rangle$ such that $\forall i \mid\mid T_i = \top$.

The concretization, union and intersection functions; and the equivalence and order relationships are lifted in an element-wise fashion from term grammars to abstract substitutions. And using the adjoint function of γ: α as the abstraction function, Θ as the domain of concrete substitutions and Ω as the domain of abstract substitutions it can be shown that $(2^{\Theta}, \alpha, \Omega, \gamma)$ is a Galois insertion [32].

We will not get into the technical details of the abstract unification operation, but it can also be found in [32].

3.1 Typeslib

As mentioned before, Typeslib is a Ciao library which gathers the operations on regular types and the main object of study of this thesis. The principal function of Typeslib is to provide basic operations for the implementations of most of the abstract domains that do regular type inference analysis in CiaoPP. To achieve this, it implements the set of basic types for analysis, the lattice that defines their order relation, and all the routines to store, manipulate, compare and operate on regular types.

This is the basic type lattice of as implemented in Typeslib ($A \rightarrow B$ means $A \subset B$):

The top type $\top$ represents any term that can be defined in Ciao, while the bottom type $\bot$ commonly appears as the intersection of two disjoint types during analysis and constitutes the empty set. As for the rest of the base types, Typeslib uses Ciao's native types as reference for its basic type lattice.

Typeslib implements type comparison, checking, and combination operations (such as least upper bound or intersection) following the algorithms given by Dart and Zobel in [4] with some performance optimizations such as tabling to avoid computing the same thing several times during analysis.

It also implements the storage of types in Ciao's database, query operations, and a large array of simplifications and cleanup operations for the database. All the type operations needed by CiaoPP's regular type analysis domains, including several widening operations

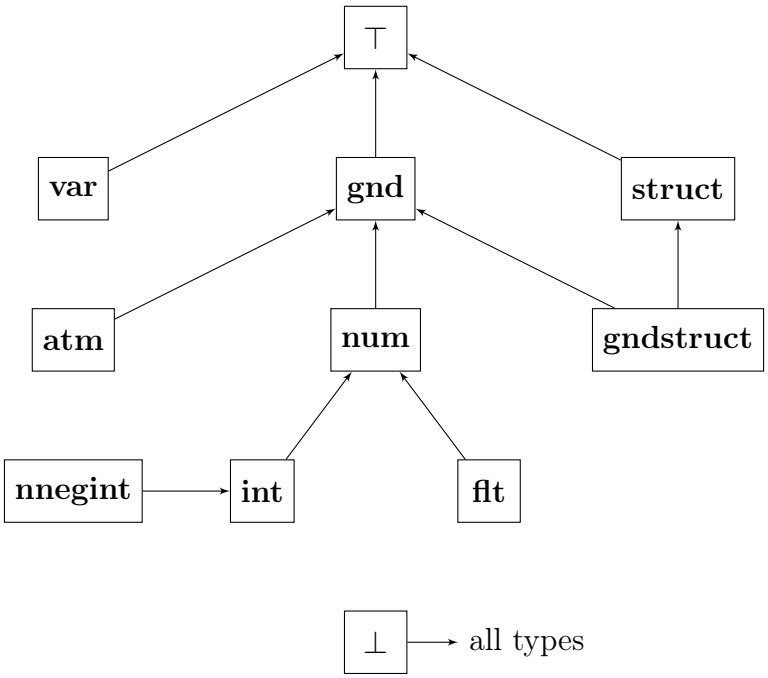

Figure 3.1: *Graph representing the basic regular types of Typeslib and their respective order relations.*

are also coded in this library.

Typeslib implements a few tools for debugging and displaying the current state of the type database too. We developed our detailed monitoring and logging tools in a fork of Typeslib to switch and activate them, as the different experiments required, using Ciao's package technology [10].

In the following sections we will present the different choices for type widening operations that are implemented in Typeslib with special attention to **structural type widening** and **defined types widening**.

3.1.1 Widening Operations

Perhaps the two most important families of operations for regular type analysis are the widening and simplification operations. These operations are key to the efficiency and precision of the type analysis.

The basic dynamic of type analysis is the derivation of regular types until a fix-point is

reached for the program: the abstract interpreter derives types form the code, starting from the base cases and iterating over clauses with the already derived suitable types as inputs until no new types appear and a fix-point is reached.

However it is easy to see that some very simple code examples would never reach a fix-point if we didn't refine the derivation process to tackle infinite ascending chains. Consider the following program presented in [32]:

```
list_of_lists([]).
list_of_lists([L|Ls]):-
        num_list(L),
        list_of_lists(Ls).
```

```
num_list([]).
num_list([N|Xs]):-
      number(N),
      num_list(Xs).
```

If we were to naively derive the type T for arguments of `num_list`, the base iteration would yield $T_0 \to \{[]\}$, the next one $T_1 \to \{[], [\mathbf{num}|T_0]\}$, the next $T_2 \to \{[], [\mathbf{num}|T_1]\}$ and so forth, every new iteration would yield $T_i \to \{[], [\mathbf{num}|T_{i-1}]\}$, resulting in an infinite ascending chain of types, that would never reach a fix-point.

This is where both widening and simplification operations become a key part of the analysis: widening operations are required to terminate analysis with infinite ascending chains (e.g., establishing that the type for arguments of `num_list` is $T \to \{[], T\}$) and help unify types built from correlated constants into a single type that represents them all (e.g., $\alpha_{42} \to \{4, 8, 15, 16, 23, 42\}$ could be transformed into **int**). Of course this involves a loss of precision as a trade-off for efficiency and termination of the analysis. On the other hand, simplification operations discard certain computed types to avoid rechecking them or including them in the final reports. E.g.: if we have achieved a state where $\alpha_{42} \to \{4, 8, 15, 16, 23, 42\}$ appears wherever $\alpha_{23} \to \{4, 8, 15, 16, 23\}$ appears, since $\alpha_{23} \in \alpha_{42}$ we could discard α_{23} from our analysis and save time every time we need to check all types.

The trade-offs between precision and efficiency in both the widening choice and the simplification policies make their optimal choice for a specific program one of the hardest problems in the field. There is no perfect choice for all codes and purposes.

Additionally deciding how much to generalize a type when widening is a very difficult problem due to the lack of landmarks. E.g.: when doing real number interval analysis one could establish certain numbers as references and widen derived intervals to the smallest interval built with the reference numbers that contain it; this proves much more complicated for type derivation, and each widening operator implements different strategies to circumvent this problem.

These are the reasons why Typeslib implements several widening operations and simplification functions to be used by the different analysis domains. We will now briefly present the implemented widening operations following the descriptions in [32]. To illustrate them, we will also use the example program presented before regarding the type derivation for arguments of `num_list`.

Functor Widening: [20] creates a type and a production for each functor symbol in the original type. Then, all arguments of the function symbols are replaced with the original types. E.g.: in our `list_of_lists` program, the ideal typing for arguments of that predicate would be: $T_{ll} \rightarrow \{[], [T_l|T_{ll}]\}$ where $T_l \rightarrow \{[], [\mathbf{num}|T_l]\}$. Using functor widening, we would reach the following, less precise fix-point: $T \rightarrow \{[], \mathbf{num}, [T|T]\}$.

Type Jungle Widening: A type jungle is a term grammar where each functor always has the same arguments. It was proposed in [16] as a finite type domain because a domain where all grammars are type jungles makes ascending chains finite. This property makes them an intuitive widening operator. E.g.: applying this operator to T_{ll} we get $T \rightarrow \{[], [T_1|T]\}$ where $T_1 \rightarrow \{[], \mathbf{num}, [T_1|T]\}$

Shortening: Treats types seen as their graph representation where nodes correspond to non-terminal grammar symbols (or-nodes) and to the right hand sides of their productions (and-nodes). The edges represent a production relationship or an argument being used by a right hand side functor.

Given an or-node (circle) its **principal functors** are the functors appearing in its children nodes (squares). The Shortening widening defined in [9] avoids having two or-nodes,

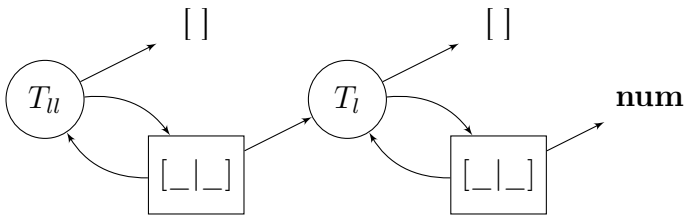

Figure 3.2: *Graph from [32] representation of the grammar* $T_{ll} \rightarrow \{[], [T_l|T_{ll}]\}, T_l \rightarrow \{[], [\textbf{num}|T_l]\}$.

connected by a path with the same principal functors (in this case T_l and T_{ll}). In the case two such nodes are found, they are replaced by their least upper bound; yielding for this example: $T \rightarrow \{[], [T_1|T]\}, T_1 \rightarrow \{[], \textbf{num}, [\textbf{num}|T]\}$.

Typeslib's implementation of shortening adds an additional depth constraint, where it only checks if two nodes are connected up to the specified depth. This contrasts with the original shortening definition, where the length of a connection is disregarded.

Depth Widening (*depth-k*): Proposed in [14], prunes the type depth after reaching the specified value (commonly called *depth-k*); additionally the analysis works exclusively with deterministic types. Therefore the result of the analysis depends on the maximum allowed depth, achieving higher precision for higher values. E.g.: pruning the widening of T_{ll} at $depth = 1$ would yield $T \rightarrow \{[], \textbf{num}, [T|T]\}$; while pruning at $depth = 2$ would yield $T \rightarrow \{[], [T_1|T]\}, T_1 \rightarrow \{[], \textbf{num}, [T_1|T]\}$.

3.1.2 Structural Type Widening

The previously described widening operators implemented in Typeslib looked for child nodes with the same functors and generalized them with a recursive structure. This allowed for fast fix-point achievement, but introduced a loss of precision that could, in the worst cases, derive recursive types for non-recursive structures with similar functors at different point.

The principal contribution of [32] was the introduction of **Structural Type Widening**, which is the default widening of the Eterms domain (that will be presented in 3.2.1), the main case of study of this thesis.

Structural type widening focuses on the program structure to distinguish between recursive and non-recursive types. It employs **type names** to identify each variable in each argument of each variant of each program atom of each predicate. This preserves information on how types are formed from other types during analysis and detects recursion easily.

When a new type approximation for a type name is made, structural widening computes their least upper bound and looks at sub-terms of that approximation which reference the original type name to make them explicit in their own new types. Structural widening is correct and very precise, but this comes with a high computational cost (least upper bound operations, their parsing, new type generation, cross referencing type names...) and the possibility of non-termination in some contrived cases where new approximations to a type name do not contain references to the previous approximation.

In the Typeslib implementation, termination is ensured by a bound on the maximum number of approximations allowed for each type name.

3.1.3 Defined Types Widening

Another kind of widening implemented in Typeslib is the *defined types* widening, which is not designed to be used on each iteration to ensure that a fix-point is reached eventually like the others. In contrast, it is only used once, after several iterations to reach a fix-point in one step.

Defined types widening chooses the closest generalization of the current approximation of a type from a pre-compiled list of defined target types (typically the regular type definitions that are visible in the module being analyzed). E.g.: Imagine that our list of defined types is $T_0 \rightarrow \{[], \mathbf{num}, [T_0|T_0]\}$ and $T_1 \rightarrow \{[], \text{“0”}, \text{“1”}, [T_1|T_1]\}$ then the widening of $S_0 \rightarrow \{[], \text{“0”}, \text{“2”}, \text{“4”}, [S_0|S_0]\}$ would be T_0, while the widening of $S_1 \rightarrow \{[], \text{“1”}, [S_1|S_1]\}$ or $S_2 \rightarrow \{[], [\text{“1”}, \text{“0”}, \text{“1”}], \text{“0”}, [\text{“1”}, \text{“1”}, \text{“1”}, \text{“0”}]\}$ would be T_1.

The main use for this widening operator is to generalize type structures which are overly

specific and reduce them to known, simple, and general structures. Its precision loss and cost are determined by the number of defined types to consider. Automatically compiling this list of references for good performance-precision compromised is a challenge that can reap valuable rewards. In section 3.2.2 we will present how the Deftypes domain works and we will propose an alternative widening operator based on defined types in 5.4.

3.2 Type Analysis Domains

One of the most interesting and important features of the CiaoPP abstract interpreter for analysis is its modular framework. Ciao implements a multipurpose abstract interpreter that its able to perform analysis based on multiple abstract domains (some of them concurrently).

The idea is that any developer can write their own abstract domain, define their key operations for the abstract interpretation process (like abstraction, abstract unification, or widening), and hook their domain to any of the fix-point algorithms of CiaoPP.

Regarding type inference analysis, CiaoPP implements different abstract domains that can be divided into two different categories: the ones that carry type names to direct the analysis with structural information and the ones that do not and focus on local references for analysis. Inside each category, the main difference among the various implemented domains is not in their actual abstract domain (which is the one defined in Typeslib with or without type names) but in the widening operations that they use.

3.2.1 Eterms Domain

The Eterms type analysis domain is the default abstract domain of CiaoPP for the inference of regular types. It uses structural type widening to achieve fix-points and therefore includes type names in its abstract domain (this is explained in detail in 3.1.2).

Eterms is the most precise type analysis domain implemented in CiaoPP, and generally is the best option to apply, since the cost of running it is pretty manageable for most programs. However, there are also some bad cases where Eterms time to reach the fix-point

grows exponentially with the number and complexity of the types derived and the type unification and simplification operations do not help reduce these quantities.

Even with Eterms termination being guaranteed by a bound on the maximum number of approximations computed for each singular type name, in some of these contrived cases we need to stop the analysis and switch to a different, less precise domain. Finding out how to best proceed in these cases, to get the best possible performance-precision compromise, was one of the main problems tackled by this thesis.

Finally, Eterms performs a top-down analysis: it starts deriving types for the entry points to the program implemented and proceeds down the clauses until it reaches the bottom facts of the program analyzed. The structural type widening keeps the information about the paths from the top to the current point of analysis in the form of the type names. This helps detect and reflect any recursive structure found along the way in the analysis.

3.2.2 Deftypes Domain

The other relevant type analysis domain for our thesis is Deftypes. Deftypes performs one of the most efficient top-down analysis implemented in the CiaoPP system. It achieves this efficiency by skipping widening operations between iterations while looking for fix-points in the type approximations.

Before trying to infer the types of the analyzed module, Deftypes compiles a list of target types. This can be the set of types defined by the user or imported from libraries used by the module being analyzed, i.e., the type definitions that are visible to that module, which typically includes the basic types of the standard libraries such as lists, trees, etc. Alternatively, such a set of types can be inferred a priori from the data structures explicitly defined in the code using a different type analysis.

Deftypes iterates type approximations until it reaches a fix-point or until the maximum number of iterations allowed is reached with no widening operations in between. If no fix-point is reached within the iteration limit, Deftypes applies the defined types widening

operation defined in 3.1.3 to the latest approximation and proceeds with the analysis.

In the case that a fix-point is reached before the iteration limit is surpassed Deftypes takes the fix-point approximation as the inferred type and adds it to the list of target types for later use in the analysis.

This concept practically eliminates the cost of widening operations from the analysis, but takes its toll on the precision of the analysis. The general target types may be more or less useful for the current program under inspection and the user defined structures included are almost guaranteed to be relevant to the analysis, but Eterms is generally considered as much more precise. It is interesting to note that, except for the fact that it can still "invent" some new types on the fly, Deftypes is closest to the behavior of strongly typed systems, where all the types considered have to be defined a priori, i.e., all types considered are the ones in the list of defined types.

In section 5.4 we will present the idea of a new analysis domain that carries the main concepts of Deftypes but changes the way in which the target types list is compiled.

Chapter 4

Identifying the Bottlenecks in Eterms

This thesis aims to audit Typeslib and find ways to make type inference analysis more efficient. Eterms is the default and most precise analysis for inference of regular types implemented in CiaoPP; because of that, we made it the center of our investigation.

We knew that the regular type manipulation operations implemented in the Typeslib bundle had been coded “to work" and not necessarily to be optimally efficient. We also knew that, before the investigation begun, several really bad cases for the Eterms analyzer had been located within Ciao’s code. Thus, we wanted to obtain better knowledge of these issues and pinpoint the principal weaknesses and points of failure.

The inference of regular types in the CiaoPP environment is very complex and entails several interacting components. The three main ones are CiaoPP’s abstract interpreter, the analysis domain implementation, and the different regular types operations (including the chosen widening operation).

All of these components are subject to present bugs or improvement opportunities such as optimizations or better precision. However, when faced with the daunting task of tackling Eterms analysis we needed to perform a *triage* and see where our efforts would be best invested.

CiaoPP’s abstract interpreter is a piece of software that has been constantly looked at, updated and developed for years. It is also hooked to a huge variety of analysis domains and revisited every time a new one needs to be connected. On the other hand, Typeslib

is a very old bundle[1] written in the 90's and that has undergone an accident where some code was lost in the middle of a refactoring process. Moreover, since its development, other libraries for regular types operations such as Libvata [15] have been published, using different representations that improve the efficiency of Dart and Zobel's algorithms that Typeslib implements.

Given these factors, we deemed that Typeslib was the most probable cause of failure and inefficiency: Any changes to CiaoPP's abstract interpreter would affect the other analysis domains already implemented in CiaoPP and that new relevant technologies had been published since the implementation of Typeslib. Therefore it seemed clear that we should focus our audit on this bundle.

4.1 Checking Correctness of Basic Type Operations

The first step in our audit was to inspect the basic regular type operations implemented in Typeslib such as emptiness checks, type inclusion, type intersection, etc.. These are the building blocks of Eterms and every other regular type inference analysis domain in CiaoPP.

Although we were fairly confident in the correctness of the operations (provided that their database and argument preconditions were satisfied) we wanted more evidence to support our convictions.

The implementation of these operations comes from a Ciao encoding of the ones presented by Dart and Zobel in [4]. We decided to take a property-based testing approach: each operation result was tested as to whether or not it satisfied some properties it should. E.g.: If type C is computed as the intersection of A and B, then properties like $C \subseteq A$, $C \subseteq B$, $C \cup A = A$ $C \cap A = C$, etc. should be satisfied.

To build our test cases we built two regular type set generators. Both generators took as inputs: the number of types to generate, a list of basic types, a list of functors (with their arity) and a bound on the maximum number of clauses a type was allowed to have. Then,

[1] Library in the jargon of the Ciao language.

they used these elements to build regular types for testing. One generator was deterministic and employed a failure-driven loop to generate each possible type combination in order. The second one was stochastic, and able to generate any possible type combinations (within the parameters stipulated).

The tests were performed in the following way: First, the parameters for type generation were set. Then, the process of cleaning up the type database, generating a set of regular types, loading them into the database, performing the type operations test on the first two types of the set, printing the final state of the type database, and manually inspecting the results are iterated *ad libitum*.

This approach had its limitations, which are not minor. Most of the complex operations relied on others to work. We tried to choose a testing order for operations that avoided the use of untested operations. In some cases, however, two operations referenced each other in some of their clauses (mutual recursion). Moreover, some of the properties we tested for encountered the same problem: we had to use untested operations to check incorrectness of other operations.

This entailed the problem of having to manually review test results to confirm general good behavior and investigate unexpected answers. A more formal approach to testing would have been better but we didn't think the time investment was worth it for this purpose, as we already had good confidence in the general good behavior of these operations.

The other mayor limitation is that the database was flushed between each test case and the next, therefore limiting our capability to test failures or errors rooted in bad database management. We tried to circumvent this by logging the database state before and after cleaning it and looking for strange entries.

However serious these limitations were, the tests proved their effectiveness by diagnosing undocumented preconditions, finding a database management bug, and confirming the results of [19] of the general incorrectness of Dart and Zobel's inclusion check algorithm for tuple distributive regular types.

4.1.1 Diagnostic Tools Developed

All the diagnostic tools developed for this thesis where coded in Ciao an integrated into various forks of the Typeslib bundle.

For these tests we developed a deterministic and a stochastic version of a regular type generator, all the property checkers for the basic unary and binary regular type operations, and a module to easily perform the testing and display the results.

The type generation algorithm was first implemented as a failure-driven loop to explore all combinations in the first stages of testing. This made debugging easier, as we were always dealing with the same test cases in the same order. However, to quickly test complex instances we implemented a stochastic type generator which we describe now.

The stochastic type generator first generates as many names as the number of types to generate, then parses the input basic types and functors, cleans invalid entries, and then proceeds to generate clauses for the types referencing other types or functors. This is its main function:

```
1
2 % E.g., gentypes(2,3,[int,atm,flt,rtwe],[e,a/3,df/e,r/0,atm/5,a/1,f/3],D).
3
4 gentypes(NumTypes,MaxClauses,BasicTypes,Functors,Definitions) :-
5     % Generates names for the generated types: t1, t2...
6     get_atom_names('t',0,NumTypes,Types),
7     % Cleans invalid basic types and functors passed as arguments to
8     % compile the possible clauses list.
9     get_free_types(Types,BasicTypes,AllTypes),
10    get_all_clauses(AllTypes,Functors,AllClauses),
11    % Finally generates the set of types
12    repeat,
13    once(get_type_definitions(Types,MaxClauses,AllTypes,AllClauses,'nil',
    'nil',Definitions)).
```

Listing 4.1: *Main predicate of the stochastic type generator*

Types are generated one at a time. Types have a maximum number of clause slots, for each slot an RNG[2] decides with 50/50 odds whether a clause is added or left blank. The only exception is the last slot, which is always defined. This results in the expected number of

[2]Random Number Generator

clauses being determined by a binomial distribution.

After the first two types are defined in this fashion, they are removed form the possible clause list to ensure that they are entry points to the type database. We keep track of the generated (or to be generated) types referenced in the clauses of the already defined types; if an unreferenced type were to be generated, instead of that, it is skipped and removed form the list of possible clauses. This improves the efficiency of the generator while not reducing its range of outputs. The predicates that make this possible are presented below:

```
1
2  % Iterates through all types to define.
3  get_type_definitions([],_,_,_,_,_,[]):-!.
4  get_type_definitions([T|Types],MaxClauses,AllTypes,AllClauses,L1,L2,[D|
       Definitions]) :-
5      f_check(L1,L2,T),!,
6      random(1,2,X),
7      get_type_definition(X,T,MaxClauses,AllTypes,AllClauses,D,Ts),
8      sort(Ts,TT),
9      f_update(L1,L2,(T,TT),L1_1,L2_1),
10     get_type_definitions(Types,MaxClauses,AllTypes,AllClauses,L1_1,L2_1,
       Definitions).
11 get_type_definitions([T|Types],MaxClauses,AllTypes,AllClauses,L1,L2,
       Definitions) :-!,
12     select(T,AllTypes,AllTypes1),
13     select(T,AllClauses,AllClauses1),
14     get_type_definitions(Types,MaxClauses,AllTypes1,AllClauses1,L1,L2,
       Definitions).
15 get_type_definitions(_,_,_,_,_,_,_,[]).
16
17 % Iterates through all clauses defined in a type
18 get_type_definition(_,_,_,_,[],_,_):-!.
19 get_type_definition(_,Type,1,AllTypes,AllClauses,[Definition],Ts) :-!,
20     random_select(Clause,AllClauses,_),
21     def_this_thing(Type,Clause,AllTypes,Definition,Ts).
22 get_type_definition(1,Type,Clauses,AllTypes,AllClauses,Definition,Ts) :-
23     Clauses > 1,
24     Clauses1 is Clauses -1,
25     random(1,2,X),
26     get_type_definition(X,Type,Clauses1,AllTypes,AllClauses,Definition,Ts)
       .
27 get_type_definition(2,Type,Clauses,AllTypes,AllClauses,[D|Definition],Ts1)
       :-
28     random_select(Clause,AllClauses,AllClauses1),
29     def_this_thing(Type,Clause,AllTypes,D,T),
30     append(T,Ts,Ts1),
31     Clauses > 1,
32     Clauses1 is Clauses -1,
33     random(1,2,X),
```

```
34      get_type_definition(X,Type,Clauses1,AllTypes,AllClauses1,Definition,Ts
        ).
35
36
37  % Auxiliary predicates to keep track of referenced and defined types.
38  f_check('nil','nil',_):-!.
39  f_check((_,_),'nil',_):-!.
40  f_check('nil',(_,F),T):-member(T,F).
41  f_check((_,F1),(_,F2),T):-append(F1,F2,F),member(T,F).
42
43  f_update('nil','nil',T,T,'nil'):-!.
44  f_update('nil',(ID,F),(T,TT),'nil',(ID,U)):-member(T,F),!, union(F,TT,U).
45  f_update((ID,F),'nil',(T,TT),'nil',(ID,U)):-member(T,F),!,union(F,TT,U).
46  f_update((ID,F),'nil',(T,TT),'nil',(T,U)):-member(ID,TT),!,union(F,TT,U).
47  f_update((ID,F),'nil',(T,TT),(ID,F),(T,TT)):-!.
48
49  f_update((_,[]),(ID,F),(T,TT),'nil',(ID,U)):- member(T,F),!, union(F,TT,U)
        .
50  f_update((ID,F),(_,[]),(T,TT),'nil',(ID,U)):- member(T,F),!, union(F,TT,U)
        .
51
52  f_update((ID1,F1),(ID2,F2),(T,TT),'nil',(ID1,U)):-
53      member(T,F1),member(ID2,TT),!,union(F1,TT,U1),union(U1,F2,U).
54  f_update((ID1,F1),(ID2,F2),(T,TT),(ID1,U),(ID2,F2)):-
55      member(T,F1),!,union(F1,TT,U).
56  f_update((ID1,F1),(ID2,F2),(T,TT),'nil',(ID2,U)):-
57      member(T,F2),member(ID1,TT),!,union(F2,TT,U2),union(U2,F1,U).
58  f_update((ID1,F1),(ID2,F2),(T,TT),(ID1,F1),(ID2,U)):-
59      member(T,F2),!,union(F2,TT,U).
```

Listing 4.2: *Type definition clause generation predicates*

Each defined type, basic type and functor has the same chance of being chosen as a clause. If a functor was chosen, we restricted its arguments to be types symbol and did not allow for them to be other functors; additionally we forbade different functors from appearing as other clauses for the same type. These two restrictions ensure that the types generated are regular, not tuple-distributive and in normal form. The rest of the predicates of the module, including the clause generation ones are not very interesting and therefore omitted.

The testing algorithm was explained in the previous section: The type database is cleaned up and a new set of regular types is generated. Once loaded into the database, we use them to check type operations, and the entire process is logged and display for human analysis. A simplified version of the main testing function is presented below.

```
% E.g., type_op_test(7,3,[int,flt,atm],[f/3,g/2]).
type_op_test(NumTypes,MaxClauses,BaseTypes,Functors) :-
    % Clean type database
    extra_cleanup_types,
    % printing the asserted facts after cleanup
    list_type_asserts(_),
    % Generate the types
    gentypes(NumTypes,MaxClauses,BaseTypes,Functors,Definitions),
    % Load types to the database
    insert_types(Definitions),
    post_init_types,
    % Printing asserted facts after loading types
    list_type_asserts(_),
    % Perform operation tests on the first two types
    test_pair((t1,t2)),
    % Display final state of the database
    pretty_print_type_definitions(Definitions),
    show_types,
    % Read command line for halt or new test
    get_line(X),
    end(X).

end("1").
```

Listing 4.3: *Main loop for the correctness test of type operations.*

From all of the generated types, we only test operations on the two first ones. The others are useful to have as context and to build more complex types that reference others. We only test the basic regular type operations of Typeslib, the building blocks for the rest of the module.

First we test the unary operations to check if the types are empty, ground, infinite, $\top$, or numeric. Then we check the binary operations of inclusion checking, union and both intersection implementations[3]. For binary operations, both argument orders are tested and the results are logged all across the board.

```
test_pair(Pair) :-
    test_empty_type(Pair),
    test_is_ground_type(Pair),
    test_is_infinite_type(Pair),
    test_equivalent_to_top_type(Pair),
    test_equivalent_to_numeric(Pair),
    test_dz_type_included(Pair),
```

[3]The main difference between `type_intersection_0/3` and `type_intersection_2/3` is that the first one just computes the intersection and outputs it, while the second one looks in the database for an equivalent type to the computed intersection before answering it.

```
    test_union_pairs(Pair),
    test_intersect_pairs_0(Pair),
    test_intersect_pairs_2(Pair).
```

Listing 4.4: *Testing function for unary an binary regular type operations over a pair of regular types.*

We present the testing code for the union operation to illustrate how the rest of tests where implemented.

```

test_union_pairs((P1,P2)) :-
    % Making sure that the preconditions are satisfied for good behaviour
    resetunion, %(necessary even after resetting the database.)
    % Call to typeslib operation to check
    type_union(P1,P2,U),
    % If sucessful, log it and check properties.
    display_color(blue, union(P1,P2,U)),
    union_properties(P1,P2,U).

union_properties(P1,P2,U):-
    union_p1(U,P1),
    union_p1(U,P2),
    union_p2(U,P1),
    union_p2(U,P2),
    union_p3(U,P1),
    union_p3(U,P2),!,
    % If all properties are satisfied we log it in green
    display_color(green,'All good with the union.').

% If some property fails, we log it in red
union_properties(_,_,_):- display_color(red,'Something went wrong in the union.').


% LIST OF UNION PROPERTIES TO CHECK
% Union is ge than both original types
union_p1(U,T) :-
    dz_type_included(T,U),
    display_color(blue,T =< U).

% The intersection of a union with one of their components is that component
union_m2(U,T) :-
    type_intersection_2(U,T,X),
    dz_equivalent_types(T,X),
    display_color(blue,T = T /\ U).

% Union is invariant to the union with an original type
union_p3(U,T) :-
    resetunion,
```

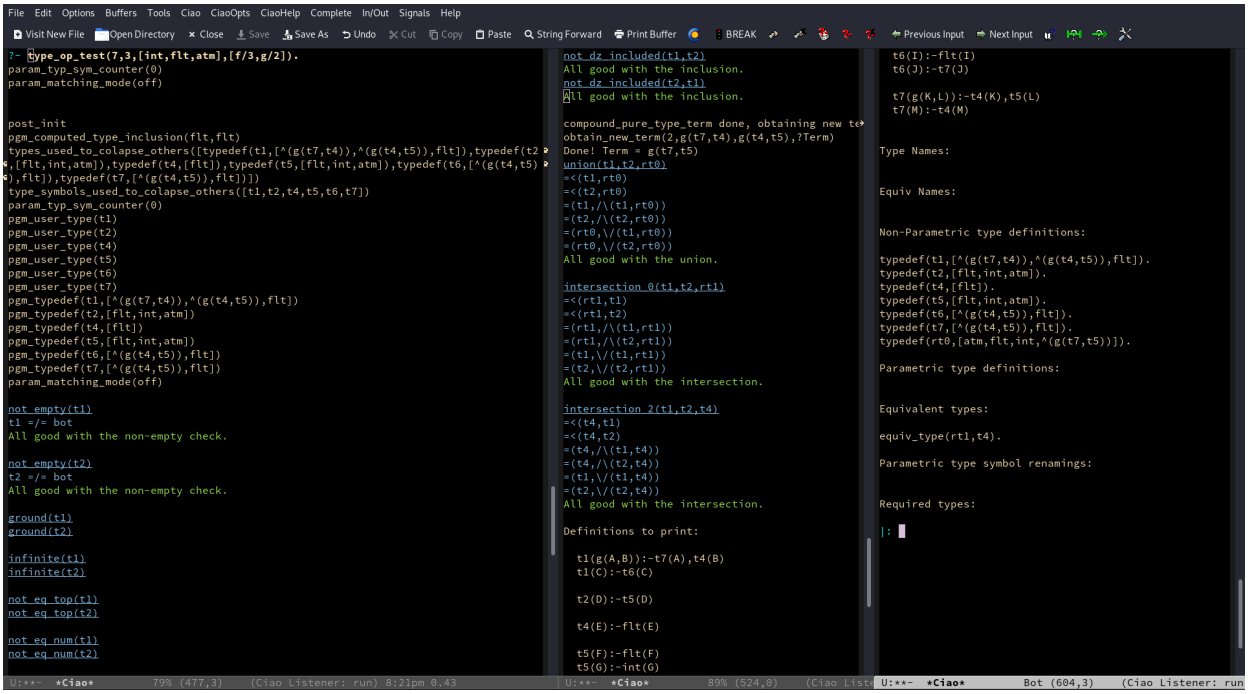

Figure 4.1: *Output of a stochastic property-based basic regular type operation test using the code of Listing 4.3*

```
40      type_union(U,T,X),
41      dz_equivalent_types(U,X),
42      display_color(blue,U = T \/ U).
```

Listing 4.5: *Implementation of the test on the union operation for regular type.*

Our logging color scheme was cream for database queries, blue for analysis results, green for successful operation checks and red for unexpected bad behavior. The result of a random test is displayed in Figure 4.1.

4.1.2 Results Obtained

Although with these tests we cannot prove that the basic type operations work perfectly and are correct under their preconditions, we found no evidence of them being incorrect for non tuple-distributive types. This, plus the general precision and correctness of type inference analysis that use these operations, justify our reasonable conviction of the tested type operations behaving as expected.

During the development of the testing tools and the early testing process a few bugs were diagnosed and corrected regarding database assertion management, mostly in the cleanup operations. These findings led to the development of the `extra_cleanup_types/0` predicate (Listing 4.3 line 4) to actually wipe the database when the original `extra_cleanup_types/0` predicate did not. In a similar manner, we were able to detect an undocumented precondition on the `type_union/3` operation: that it required calling `reset_union/0` before it run to work properly. Finally, while developing the operation property tests we could find some incorrect answers for tuple distributive type declarations, as shown in [19]. This was expected and accounted for in the development of Typeslib.

We could also confirm that the two type intersection operations `type_intersection_0/3` and `type_intersection_2/3` worked as intended. In Figure 4.1 the analysis shows how the first version `type_intersection_0/3` derives a new type `rt1` for the intersection. This new type is then stored in the database. It also shows how `type_intersection_2/3` answers an equivalent type `t4`, sets the equivalence in the database and removes the definition of `rt1`.

We could also observe no errors in the simplification process from type definitions in declaration to their non-parametric definitions computed by Typeslib. However, as shown by the non-parametric definitions of `t6` and `t7` in Figure 4.1, those types could have been further simplified to achieve a smaller database. This is a consequence of a trade-off between memory and time consumption of the simplification operations made in the development of Typeslib. Any time we want to simplify equivalent types we have to check all type pairs in the database for equivalence; if one is detected, store an equivalence relation, unify all references in the stored type definitions and re-start the process for new equivalence that these unifications have made visible. The cost of this kind of simplifications is large and therefore the Ciao Development Team chose to have redundant information in memory.

4.2 Measuring Calls and Costs

From our previous tests, we could find and fix some problems of the Typeslib library, mainly with operation preconditions and database assertion management. However, none of the detected problems related to the inefficient behavior of Eterms for problematic cases.

To proceed with our investigation, we decided to focus our efforts on a specific module for which Eterms analysis was known to take a long time. The chosen module also had to be simple and easily adaptable; that way, we would be able to edit it to perform different experiments and have a wider range of movement while exploring the problem.

The chosen problematic module, extracted from Ciao's `lpdoc` bundle is presented below.[4] To perform different tests with this module, some of the different clauses of the `html_escape/2` predicates where commented and others left active. We decided to run experiments, from the bottom clause up by progressively activating clauses, if a clause addition resulted in an analysis taking longer than ten minutes, it would be commented and skipped for the following analysis. The numbers commented to the right of the clauses represent in which tests where the clauses active.

```
1  :- module(_,[html_escape/2],[]).
2
3  %html_escape("‘‘"||S0,"&ldquo;"||S):- !, html_escape(S0, S).%7 + this was
       too slow
4  %html_escape("’’"||S0,"&rdquo;"||S):- !, html_escape(S0, S).%7 + this was
       too slow
5  html_escape([34|S0], "&quot;"||S) :- !, html_escape(S0, S). %7
6  html_escape([39|S0], "&apos;"||S) :- !, html_escape(S0, S). %6,7
7  html_escape("&#"||S0, "&#"||S) :- !, html_escape(S0, S).    %5,6,7
8  %html_escape([38|S0], "&amp;"||S) :- !, html_escape(S0, S). %4 + this was
       too slow
9  html_escape([60|S0], "&lt;"||S) :- !, html_escape(S0, S).   %4,5,6,7
10 html_escape([62|S0], "&gt;"||S) :- !, html_escape(S0, S).   %3,4,5,6,7
11 html_escape([X|S0], [X|S]) :- !, html_escape(S0, S).        %2,3,4,5,6,7
12 html_escape([], []).                                        %1,2,3,4,5,6,7
```

Listing 4.6: *Problematic module for Eterms analysis with some clauses commented out. The code is prepared to run the seven active clauses test, the biggest configuration able to finish Eterms analysis in less than ten minutes.*

[4] This module translates between the internal and html encoding of different special characters or strings.

When running our experiments we wanted to record the time taken in the different steps of the Eterms analysis; namely the set up, abstract interpretation and output preparation. We also wanted to be able to monitor type operations, their number of calls and the total time spent performing them. This way we would get better insight on where most of the time was being spent and which operations to optimize.

4.2.1 Diagnostic Tools Developed

The main tool developed for this analysis was the Ciao package `record_call`. In Ciao, packages allow for automatic rewriting of the code of a module before compilation. This has many uses, but in this case it allows us to declare at the beginning of a module a list of predicates to monitor and to add to the call records database. Before compilation of the module, Ciao uses the package to rewrite the code of the declared predicates adding to them all the recording artifacts needed.

```
:- use_package(library(record_call)).

:- use_record_call(dz_type_included/2).
:- use_record_call(edz_type_included/5). % for sized types
:- use_record_call(dz_equivalent_types/2).
:- use_record_call(is_ground_type/1).
% :- use_record_call(type_intersection_0/3). % for nfsets
:- use_record_call(type_intersection_2/3).  % for eterms domain
% :- use_record_call(is_infinite_type/1). % for non-failure
% :- use_record_call(finite_unfold/2). % for non-failure
% :- use_record_call(not_expandible_type/1). % for non-failure
:- use_record_call(equivalent_to_top_type/1).
:- use_record_call(equivalent_to_numeric/1).
:- use_record_call(is_empty_type/1).
:- use_record_call(type_symbol/1).
```

Listing 4.7: *Example showing how to use the* ***record_call*** *package in a module and the syntax for declaring a predicate to be recorded*

The package has two main files and a binder: one of the files implements all the predicates needed to record declared predicate calls, and the other implements the rewriting rules to add the recording predicates before compilation of the target module. The binder file just tells Ciao where to find the package translation rules and predicates.

As for the recording process, we decided to build a call record database, with access,

modification and display operations. We chose to keep track of the number of calls to each predicate, the total time spent in calls to that predicate and the total time spent in calls to all recorded predicates combined.

Two recording modes were considered, either we could record the calls to all declared predicates at every time they were invoked, or we could only record *top-level* calls to declared predicates: calls made from outside predicates already being recorded.

We found that the second operation mode provided clearer and more compact results, and it was the one we employed during our tests. In the following listing, the basic predicates for call recording are presented.

```
% This clause prevents recording calls inside other recorded calls
record_call(Pred,Print,Module):-
    recording(X),!,
    (call(Pred) -> Succ=true;Succ=false ),
    Succ=true.

% This clause records the call
record_call(Pred,Print,Module):-
    assertz_fact(recording(Pred)),
    statistics('walltime',[T1,_]),
    (call(Pred) -> Succ=true;Succ=false ),
    statistics('walltime',[T2,_]),
    retractall_fact(recording(_)),
    T is T2 - T1,
    functor(Print,Name,A),
    add_record(Name,A,T,Module),
    Succ=true.

add_record(FullName,A,T,M):-
    atom_concat(Name,'$_auxiliar',FullName),
    retract_fact(recorded_calls(M,Name,A,N,OT,_)), !,
    N1 is N + 1,
    OT1 is OT + T,
    Avg1 is OT1 / N1,
    assertz_fact(recorded_calls(M,Name,A,N1,OT1,Avg1)).
add_record(FullName,A,T,M):-
    atom_concat(Name,'$_auxiliar',FullName),
    assertz_fact(recorded_calls(M,Name,A,1,T,T)).
```

Listing 4.8: *Basic predicates implemented to record predicate calls.*

Translation rules look for patterns of code in the target module (first argument) and allow the programmer to make automatic modifications to those patterns (second argument). The

translation rules for the package look like this:

```
% Translation rules:

record_call_sentence_tr((Head :- Cls),New,M) :- !,
    functor(Head,Name,Arity),
    use_record_call(Name,Arity,M),
    Head =.. [Name|Args],
    atom_concat(Name,'$_auxiliar',Name1),
    NewHead =.. [Name1|Args],
    build_substitution(Head,Name,Arity,NewHead,Cls,New,M).

record_call_sentence_tr(_Fact,_,_M) :- !,  fail. % same as normal clause

record_call_goal_tr(Goal,NewGoal,M):-!,
    functor(Goal,Name,Arity),
    pred_record_call(M,Name,Arity),
    Goal =.. [Name|Args],
    atom_concat(Name,'$_auxiliar',Name1),
    NewGoal =.. [Name1|Args].

% Auxiliary functions:

build_substitution(_,Name,Arity,NewHead,Cls,[(NewHead :- Cls)],M):-
    recorded(M,Name,Arity),!.
build_substitution(Head,Name,Arity,NewHead,Cls,[Clause,(NewHead:-Cls)],M)
   :-
    assertz_fact(recorded(M,Name,Arity)),
    Clause = (Head :- record_call(NewHead,NewHead,M)).
```

Listing 4.9: *Some transaltion rules of the* ***record_call*** *package*

This is an example predicate taken from Typeslib before and after translation:

```
dz_type_included(Type1, Type2):-
    retractall_fact(pgm_dz_pair(_, _)),
    dz_subset(Type1, Type2).

dz_type_included_tabling(Type1, Type2):-
    computed_type_inclusion(Type1, Type2),
    !.
dz_type_included_tabling(Type1, Type2):-
    dz_type_included(Type1, Type2),
    asserta_fact(pgm_computed_type_inclusion(Type1, Type2)).
```

Listing 4.10: *Typeslib predicate before automatic translation by the* ***record_call*** *package.*

```
dz_type_included(Type1,Type2) :-
    record_call('dz_type_included$_auxiliar'(Type1,Type2), '
   dz_type_included$_auxiliar'(Type1,Type2),typeslib).


'dz_type_included$_auxiliar'(Type1,Type2) :-
```

```
    retractall_fact(pgm_dz_pair(_1,_2)),
    dz_subset(Type1,Type2).

dz_type_included_tabling(Type1,Type2) :-
    computed_type_inclusion(Type1,Type2),
    !.
dz_type_included_tabling(Type1,Type2) :-
    'dz_type_included$_auxiliar'(Type1,Type2),
    asserta_fact(pgm_computed_type_inclusion(Type1,Type2)).
```

Listing 4.11: *Typeslib predicate after automatic translation by the* ***record_call*** *package.*

We can see how the original predicate is changed to just call an auxiliary one with its same name and an "$auxiliar" tag that preforms the same calls as the original did. Moreover, all internal references of declared predicates to other declared predicates are changed to the "$auxiliar" ones, to prevent useless calls to the `record_call/3` predicate. The first predicate in Listing 4.8 is still needed, since a recorded predicate A may call a non recorded predicate B that calls a recorded predicate A', which we do not want to record at that moment, but we would like to record the call if no other declared predicate was being recorded at call time.

The last piece of code was the main testing function that analyses our test module. This predicate first loads the test module in CiaoPP, then uses the Eterms analyzer on it, and finally outputs the results of the analysis. For each of these stages it prints the time taken, displays the number and total time of calls to recorded predicates, and resets the record for the next stage:

```
main :-
    statistics('walltime',[T1,_]), module(lpdoc_problem),
    statistics('walltime',[T2,_]), T is T2 -T1,
    display(T), display(' ms.'), nl, nl, list_recorded_calls,
   clean_records, nl,
    statistics('walltime',[T3,_]), display('eterms'),nl,
    analyze(eterms),statistics('walltime',[T4,_]), TT is T4 -T3,
    display(TT), display(' ms.'), nl, nl, list_recorded_calls,
   clean_records, nl,
    statistics('walltime',[T5,_]), display('output'), nl, output,
    statistics('walltime',[T6,_]), TTT is T6 -T5, display(TTT),
    display(' ms.'), nl, nl, list_recorded_calls, clean_records.
```

Listing 4.12: *Main testing function for Eterms performance.*

For our initial test we decided to declare (for recording) all predicates exported by the Typeslib module. Bellow we show the results of a test with seven clauses of `html_escape/2` active (the ones labeled with the number 7 in Listing 4.6) to illustrate the output format:

```
?- main.
{Loading current module from /home/daniel/Desktop/IMDEA/ciao-devel/bndls/
   typeslib/tests/lpdoc_problem.pl
{loaded in 287.78 msec.}
}
318.58300000000054 ms.

typeslib: cleanup_types/0:
1 calls   0.03900000000066939 ms.   0.03900000000066939 ms/call
typeslib: legal_user_type_pred/1:
113 calls   0.3239999999987049 ms.    0.00286725663715668 ms/call
typeslib: insert_user_type_pred_def/2:
113 calls   1.8189999999940483 ms.    0.016097345132690694 ms/call
typeslib: post_init_types/0:
1 calls   148.90600000000086 ms.    148.90600000000086 ms/call

Total recorded time: 151.08799999999428 ms.

eterms
{Analyzing /home/daniel/Desktop/IMDEA/ciao-devel/bndls/typeslib/tests/
   lpdoc_problem.pl
{preprocessed for the plai fixpoint in 0.5820000000003347 msec.}
{analyzed by plai using eterms with local-control off in
   278.58999999999924 msec.}
}
563.4360000000015 ms.

typeslib: get_type_name/2:
14 calls  0.023999999995794496 ms.    0.0017142857139853212 ms/call
typeslib: new_type_name/1:
8 calls   0.0610000000015134 ms.    0.007625000000189175 ms/call
typeslib: dz_equivalent_types/2:
18 calls  3.029000000004089 ms.   0.16827777777800496 ms/call
typeslib: type_escape_term_list/2:
20 calls  0.11399999999412103 ms.   0.005699999999706051 ms/call
typeslib: equivalent_to_top_type/1:
40 calls  0.10200000000804721 ms.   0.0025500000000201804 ms/call
regtype_basic_lattice: set_top_type/1:
55 calls  0.03800000000774162 ms.   0.0006909090910498476 ms/call
typeslib: type_intersection_2/3:
39 calls  0.23599999999896681 ms.   0.0060512820512555595 ms/call
typeslib: retract_type_name/3:
47 calls  0.10099999999692955 ms.   0.002148936170147437 ms/call
typeslib: insert_type_name/3:
55 calls  4.246999999988475 ms.   0.07721818181797227 ms/call
typeslib: normalize_type/2:
```

```
39 calls  0.3109999999978754 ms.    0.007974358974304497 ms/call
regtype_basic_lattice: top_type/1:
48 calls  0.030000000004292815 ms.    0.0006250000000894337 ms/call
typeslib: resetunion/0:
24 calls  0.13100000000304135 ms.   0.0054583333334600566 ms/call
typeslib: type_union/3:
24 calls  16.652000000000044 ms.    0.6938333333333352 ms/call
typeslib: lnewiden_el/4:
24 calls  258.491 ms.   10.770458333333332 ms/call
typeslib: get_canonical_name/2:
289 calls   0.3150000000005093 ms.    0.0010899653979256378 ms/call

Total recorded time: 283.8820000000014 ms.

output
He llegado a simplify_step2
He salido de simplify_step2 en 3416.3379999999997ms.
{written file /home/daniel/Desktop/IMDEA/ciao-devel/bndls/typeslib/tests/
   lpdoc_problem_eterms_co.pl}
3432.5869999999977 ms.

typeslib: simplify_step2/0:
1 calls   3416.3730000000014 ms.    3416.3730000000014 ms/call
typeslib: revert_type_internal/3:
4 calls   0.008000000001629815 ms.    0.0020000000004074536 ms/call
typeslib: revert_types/5:
2 calls   0.0319999999992433 ms.    0.01599999999962165 ms/call
typeslib: get_required_types/1:
1 calls   0.010999999998603016 ms.    0.010999999998603016 ms/call

Total recorded time: 3416.424000000001 ms.
```

Listing 4.13: *Sample output of the test Eterms analysis of the html_escape module with seven active clauses (test 7).*

4.2.2 Results Obtained

Our test battery consisted of 50 runs of each test configuration (from 1 to 7 in Listing 4.6) and 3 runs of the combinations that were too slow to finish in ten minutes: test 4 and the clause in line 8, test 7 and the clause in line 3 and test 7 and the clause in line 4. The slow runs were eventually aborted after ten minutes of execution.

As we saw in Listing 4.13 most of the time for each stage was recorded, most of the time during the Eterms analysis was consumed by `lnewiden_el/4` and most of the type at the output generation stage was consumed by `simplify_step2/0`. That is why we decided

to have those values as our main benchmarks for the tests. The average times and their standard deviation are shown below:

Eterms Benchmarks								
Test	Load	R.Load	Eterms	R. Eterms	Widen	Output	R. Output	Simplify
Test1	313.15	145.57	243.508	0.21	0	17.44	11.42	11.38
±	11.09	0.81	11.50	0.01	0	0.31	0.24	0.24
Test2	334.08	148.21	262.55	1.80	0.35	63.98	55.74	55.70
±	23.41	1.43	32.55	0.16	0.05	1.13	0.55	0.55
Test3	344.55	176.40	259.80	7.20	2.56	248.88	235.85	235.81
±	71.71	61.09	10.41	0.37	0.20	20.92	22.10	22.10
Test4	319.45	148.38	264.28	10.99	3.71	377.01	364.74	364.69
±	6.41	2.34	9.75	0.50	0.12	8.92	7.91	7.89
Test5	319.73	149.32	270.88	16.58	6.10	674.96	661.98	661.93
±	9.96	2.28	13.53	0.19	0.21	10.49	10.33	10.33
Test6	325.62	150.82	448.40	186.54	169.98	2,363.88	2,302.25	2,301.18
±	6.34	3.12	20.21	11.79	10.83	18.26	18.66	18.70
Test7	328.29	152.70	541.75	283.14	257.71	3,444.69	3,428.33	3,428.29
±	8.12	2.79	12.27	1.93	1.68	26.85	26.88	26.88

Table 4.1: *Table showing average times measured in ms. and their standard deviation. R. stands for "Recorded".*

The results obtained form this first battery of tests show that loading time barely increased with the complexity of the test module. Also, the cost of both the Eterms analysis and the output preparation ramp up by a big margin with the complexity.

Most of the cost of the Eterms analysis (all sans circa $250ms$) falls on the widening operation. And almost the entirety of the cost of output preparation is due to the simplification operations. It should be noted that these simplifications are made in order to present the inferred types in the most programmer-friendly way in the output produced by CiaoPP, but are not necessary to obtain a correct analysis output (in fact, this simplification is optional).

Additionally, all of the tests reached the most desirable conclusion in their analysis regarding precision according to the Ciao Development Team: that arguments had to be lists.

```
1 :- module(_1,[html_escape/2],[assertions,regtypes]).
2
```

```
:- true pred html_escape(_A,_B)
   : ( term(_A), term(_B) )
   => ( list(_A), list(_B) ).
```

Listing 4.14: *Conclusions of Eterms analysis of Listing 4.6 on tests 1 to 7.*

With just these tools, we had no way to accurately measure how much more complex each test was with respect to the previous one. Therefore, at this point we could not reach a conclusion on how these costs escalate with the different clauses of the test.

We knew that the simplification process was prone to suffer from combinatorial explosions in the number of types derived, as explained at the end of section 4.1.2. But we still needed to find the reason behind the cost of widening operations.

The other three tests, the ones that were aborted after ten minutes, showed similar times for the loading part of the test but where unable to finish the Eterms analysis in a reasonable time. Thus, they never reached the output preparation stage. This showed that the first bottleneck was not in the output simplification process (as Table 4.1 could have suggested) but in the analysis itself. Our next set of experiments was aimed to find out exactly which operations made Eterms widening so costly sometimes and why.

4.3 Main Bottlenecks Identified

We remind the reader that Eterms uses structural type widening between iterations to reach a fix-point during abstract interpretation, as described in Section 3.1.2. This widening does not avoid infinite ascending chains, so to ensure termination in bad cases, Ciao added an iteration counter with the objective of stopping the iterations if the limit was met.

The first test we did was to log this approximation counter per type name. We ran one of the slow analysis to see if the problem was that the iteration limit was too high or if iterations just took too long. The logs of one of such analyses are shown below:

```
?- main.
{Loading current module from /home/daniel/Desktop/IMDEA/ciao-devel/bndls/
   typeslib/tests/lpdoc_problem.pl
{loaded in 284.059 msec.}
}
```

```
303.0880000000034 ms.

typeslib: cleanup_types/0:
1 calls     0.04000000000814907 ms.        0.04000000000814907 ms/call
typeslib: legal_user_type_pred/1:
113 calls     0.29000000006635673 ms.        0.002566371682003157 ms/call
typeslib: insert_user_type_pred_def/2:
113 calls     1.774999999877764 ms.        0.015707964600688178 ms/call
typeslib: post_init_types/0:
1 calls     145.88499999999476 ms.        145.88499999999476 ms/call

Total recorded time: 147.98999999994703 ms.

eterms
{Analyzing /home/daniel/Desktop/IMDEA/ciao-devel/bndls/typeslib/tests/
   lpdoc_problem.pl
{preprocessed for the plai fixpoint in 0.43699999999989814 msec.}
pgm_type_name(name0 , [] , 0)
pgm_type_name(name1 , [] , 0)
pgm_type_name(name0 , [] , 0)
pgm_type_name(name1 , [] , 0)
pgm_type_name(name0 , [] , 0)
pgm_type_name(name1 , [] , 0)
pgm_type_name(name2 , [] , 0)
pgm_type_name(name3 , [] , 0)
pgm_type_name(name0 , [,([/(.,2)],name2)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name3)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name2)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name3)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 0)
pgm_type_name(name4 , [,([/(.,2)],name1)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2)],name4),,([/(.,2)
   ,/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 0)
```

Listing 4.15: *Logs on the Eterms counter for type name approximation before fixing it.*

The zeros that consistently appear as third arguments of `pgm_type_name/3` were supposed to display the counter for approximations performed. The counter was obviously not working (we saw that the corresponding code had been commented out), so we fixed the widening code and repeated the experiment. Please note that the unique identifier for a type name is the combination of the first and second arguments of `pgm_type_name/3`.

```
?- main.
{Loading current module from /home/daniel/Desktop/IMDEA/ciao-devel/bndls/typeslib/tests/lpdoc_problem.pl
{loaded in 241.155 msec.}
}
310.5410000000011 ms.

typeslib: cleanup_types/0:
1 calls   0.02899999999863212 ms.   0.02899999999863212 ms/call
typeslib: legal_user_type_pred/1:
113 calls   0.17499999999745341 ms.   0.0015486725663491452 ms/call
typeslib: insert_user_type_pred_def/2:
113 calls   1.0380000000004657 ms.    0.009185840707968723 ms/call
typeslib: post_init_types/0:
1 calls   93.51900000000023 ms.   93.51900000000023 ms/call

Total recorded time: 94.76099999999678 ms.

eterms
{Analyzing /home/daniel/Desktop/IMDEA/ciao-devel/bndls/typeslib/tests/lpdoc_problem.pl
{preprocessed for the plai fixpoint in 0.2669999999998254 msec.}
pgm_type_name(name0 , [] , 0)
pgm_type_name(name1 , [] , 0)
pgm_type_name(name0 , [] , 0)
pgm_type_name(name1 , [] , 0)
pgm_type_name(name0 , [] , 0)
pgm_type_name(name1 , [] , 0)
pgm_type_name(name2 , [] , 0)
pgm_type_name(name3 , [] , 0)
pgm_type_name(name0 , [,([/(.,2)],name2)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name3)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name2)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name3)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 0)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 1)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 1)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 1)
pgm_type_name(name4 , [,([/(.,2)],name1)] , 0)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 1)
pgm_type_name(name1 , [,([/(.,2),/(.,2),/(.,2),/(.,2)],name4),,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 1)
pgm_type_name(name1 , [,([/(.,2)],name1),,([/(.,2),/(.,2),/(.,2),/(.,2)],name4),,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 1)
pgm_type_name(name1 , [,([/(.,2)],name1),,([/(.,2),/(.,2),/(.,2),/(.,2)],name1),,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 1)
pgm_type_name(name0 , [,([/(.,2)],name0)] , 1)
pgm_type_name(name1 , [,([/(.,2)],name1),,([/(.,2),/(.,2),/(.,2),/(.,2)],
```

```
   name1),,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 1)
47 pgm_type_name(name0 , [,([/(.,2)],name0)] , 2)
48 pgm_type_name(name1 , [,([/(.,2)],name1),,([/(.,2),/(.,2),/(.,2),/(.,2)],
   name1),,([/(.,2),/(.,2),/(.,2),/(.,2),/(.,2)],name1)] , 2)
```

Listing 4.16: *Logs on the Eterms counter for type name approximation after fixing it.*

Not only did we confirm that the counter and the iteration bound were now working, but we also found two key facts for our investigation: each approximation was usually slower than the previous one and the unreasonable analysis times were not due to too many iterations on the same type names before reaching a fix-point; but in the time of said iterations.

To find the reason behind this increasing and eventually unmanageable cost of widening operations we used the `record_calls` package but avoided records of `lnewiden_el/4`. This way, we would log the internal calls of `lnewiden_el/4` to other predicates from Typeslib and see which ones were responsible for the cost.

We repeated test 7 with these changes and obtained a result where most of the Eterms analysis type was spent in calls to `make_determ_wide_rules/1`: the predicate that makes all type rules inferred by the widening deterministic.

We repeated the test again, applying the same philosophy and removing `make_determ_wide_rules/1` from the list of recorded predicates. Its internal predicate `make_deterministic/2`, which makes one type definition deterministic at a time, was responsible for the cost.

A type definition is deterministic when it does not contain two different compound type terms with the same functor and arity. Eg. $t \to \{t_1, f(t_2, t_3), g(t_4)\}$ is a deterministic while $t \to \{t_1, f(t_2, t_3), f(t_4, t_5)\}$ is not.

The determinization algorithm merges type terms from the definitions that share functor and arity until only one remains. It does it, by computing the union of their arguments at each position and creating new types if needed. These merges are performed sequentially. E.g.: The determinization process of $\{f(t_0, t_1), f(t_2, t_3), f(t_4, t_5)\}$:

$\{f(t_0, t_1), f(t_2, t_3), f(t_4, t_5)\}$

$\implies \{f(nt_0, nt_1), f(t_4, t_5)\}$ where $nt_0 \to \{t_0, t_2\}, nt_1 \to \{t_1, t_3\}$

$\implies \{f(nt_2, nt_3)\}$ where $nt_0 \to \{t_0, t_2\}, nt_1 \to \{t_1, t_3\}, nt_2 \to \{nt_0, t_4\}, nt_3 \to \{nt_1, t_5\}$

This implementation has problems. The first one is the some precision is lost in the determinization process. E.g.: the determinization of $\{f(t_0, t_1), f(t_2, t_3), f(t_4, t_5)\}$ presented above would include elements like $f(t_0, t_5)$. Additionally, a large amount of types is created, and these types are deep in their definitions rather than wide (due to the argument union for functors being performed sequentially by pairs. However, the main efficiency handicap comes from the repeated union operations for functor arguments.

The implementation for the type union and inclusion operations in Typeslib can become very slow when dealing with complex types, like the ones being derived by the determinization process. Before computing the union of a type, the predicate `type_union/3` checks whether its arguments are included in the other by invoking the inclusion check operation.

The inclusion check operation has significant cost, as it traverses the definition tree of the first type until the leaves are found, and for each of them traverses the entire definition of the second type checking for a leaf that includes the first one. Additionally, due to the way types are defined in the database, the algorithm is prone to checking the same leaf more than once. E.g.: Consider the type database given by:
t a very complex type such that $\{a, b, c, d\} \subset t$
$t_1 \rightarrow \{t_2, t_3\}$
$t_2 \rightarrow \{a, b, c\}$
$t_3 \rightarrow \{a, b, d\}$
t_4 to t_{400} needed to define t.
If we wanted to check $t_3 \subseteq t$, we would check $a \subseteq t$ and $b \subseteq t$ twice, covering the entire structure of t each time.

To finish making matters worse, `type_union/3` includes calls to `make_deterministic/2` which essentially make the algorithm perform the same inclusion check for sub-trees of the original definition several times.

These conclusions could only be achieved because the tools developed pointed us towards

the right direction. Typeslib is a huge bundle with a wide variety of use cases and the tests performed allowed us to look at the correct parts of the code. Additionally, white-box testing allowed us to monitor in real time the execution of the test and see at which operations the library struggled the most. The implementations of `type_union/3` and `make_deterministic/2` are displayed below.

```
":- pred make_deterministic(+Def1,+Def2):
 list(pure_type_term) * list(pure_type_term)#

@var{Def1} and @var{Def2} are two sorted lists of type terms with
compound type terms of different functor/arity. @var{Def1} is the
merge of both definitions, if two compound type terms have the same
functor/arity, both are replaced by a new compound type terms whose
arguments are the deterministic union of the formers.
"
% Base cases
make_deterministic([],[]):- !.
make_deterministic([X],[X]):- !.
% Checks if the first to type terms are equal
make_deterministic([TermType1,TermType2|Def1],Def):-
    compare(Order,TermType1,TermType2),
    make_deterministic0(Order,TermType1,TermType2,Def1,Def).

% If they are, one is redundant and we proceed without it.
make_deterministic0(=,_TermType1,TermType2,Def1,Def):-
    make_deterministic([TermType2|Def1],Def),!.
% If not equal and they both share functor/arity
make_deterministic0(_,TermType1,TermType2,Def1,Def):-
    compound_pure_type_term(TermType1,Term1,Name,Arity),
    compound_pure_type_term(TermType2,Term2,Name,Arity),
    !,
    functor(Term,Name,Arity),
    % We build their deterministic union
    obtain_new_term(Arity,Term1,Term2,Term),
    construct_compound_pure_type_term(Term,TermType),
    % and substitute them by the det. union
    make_deterministic([TermType|Def1],Def).
% If they do not share functor/arity, since the list of type terms is
% sorted, we can consider the first type term as part of the
% deterministic definition.
make_deterministic0(_,TermType1,TermType2,Def1,[TermType1|Def]):-
    make_deterministic([TermType2|Def1],Def).

% Terms with shared functor/arity are parsed argument-wise
obtain_new_term(0,_,_,_) :- !.
obtain_new_term(N,Term1,Term2,Term):-
    arg(N,Term1,Arg1),
    arg(N,Term2,Arg2),
```

```
    % to compute the union of the arguments at the same positions
    % and give the union a type symbol in the databes if needed
    type_union(Arg1,Arg2,Arg),
    arg(N,Term,Arg),
    N1 is N - 1,
    asserta_fact(uniontriple(Arg1,Arg2,Arg)),
    obtain_new_term(N1,Term1,Term2,Term).
```

Listing 4.17: *Type determinization operation implemented in Typeslib.*

```
% First, checks if it is alredy computed
type_union(Type1,Type2,Type3):-
    uniontriple(Type1,Type2,Type3),!.
% Then checks for type inclusion
type_union(Type1,Type2,Type3):-
    dz_type_included(Type1,Type2),!,
    Type3=Type2.
type_union(Type1,Type2,Type3):-
    dz_type_included(Type2,Type1),!,
    Type3=Type1.
% If the previous attempts fail, computes a new type for the union:
type_union(Type1,Type2,Type3):-
    % Choose a name and merge definitions
    new_type_symbol(Type3),
    maybe_get_type_definition(Type1,Def1),
    maybe_get_type_definition(Type2,Def2),
    merge(Def1,Def2,Def_s),
    insert_rule(Type3,Def_s),
    % Process native and non-native type symbols separately
    get_native_type_symbols(Def_s,Def_n,Def_nnat),
    union_of_native_types(Def_n,[],Def_natun),
    make_deterministic(Def_nnat,Defnew),
    % Merge the two sets of types and post-process
    merge(Def_natun,Defnew,Def),
    unfold_type_union(Type3,Def,UDef),
    SDef = UDef,
    sort(SDef,SDef_s),
    retract_rule(Type3),
    insert_rule(Type3,SDef_s),
    % assert the computation of this union in the database.
    asserta_fact(uniontriple(Type1,Type2,Type3)).
```

Listing 4.18: *Type union operation implemented in Typeslib.*

Even though Eterms only calls `make_determ_wide_rules/1` at most once per widening operation and only to make the few types directly involved in the operation (those that have been altered by the iteration) deterministic; in some cases, the cost of making one of these types deterministic becomes astronomical. However, the determinization process is needed as some type operations performed during analysis require deterministic types as a

precondition; additionally it prevents equivalent type definitions from proliferating in the database wasting memory space.

```
{Loading current module from /home/daniel/Desktop/IMDEA/ciao-devel/bndls/typeslib/tests/lpdoc_problem.pl
{loaded in 219.36 msec.}
}
460.3110000000015 ms.

typeslib: cleanup_types/0:
1 calls   0.03199999999196734 ms.   0.03199999999196734 ms/call
typeslib: legal_user_type_pred/1:
113 calls   0.16499999996449333 ms.   0.0014601769908362241 ms/call
typeslib: insert_user_type_pred_def/2:
113 calls   1.093999999910011 ms.   0.009681415928407176 ms/call
typeslib: post_init_types/0:
1 calls   81.37100000001374 ms.   81.37100000001374 ms/call

Total recorded time: 82.66199999988021 ms.

eterms
{Analyzing /home/daniel/Desktop/IMDEA/ciao-devel/bndls/typeslib/tests/lpdoc_problem.pl
{preprocessed for the plai fixpoint in 0.25199999999949796 msec.}
------------------------------------------------------------
I= [rt7]
------------------------------------------------------------
Working on rt7 defined as: [[],[38|rt7]]
Calling make_deterministic(Def=[[],[38|rt7]], NewDef)...
Done make_deterministic(Def=[[],[38|rt7]],NewDef=[[],[38|rt7]])
in 0.0020000000076834112ms.

------------------------------------------------------------
I= [rt9]
------------------------------------------------------------
Working on rt9 defined as: [[],[38,97,109,112,59|rt9]]
Calling make_deterministic(Def=[[],[38,97,109,112,59|rt9]], NewDef)...
Done make_deterministic(Def=[[],[38,97,109,112,59|rt9]],NewDef
   =[[],[38,97,109,112,59|rt9]])
in 0.0029999999969732016ms.

------------------------------------------------------------
I= [rt32]
------------------------------------------------------------
Working on rt32 defined as: [[],[38|rt7],[rt13|rt32]]
Calling make_deterministic(Def=[[],[38|rt7],[rt13|rt32]], NewDef)...
Done make_deterministic(Def=[[],[38|rt7],[rt13|rt32]],NewDef=[[],[rt13|
   rt32]])
in 0.13700000000244472ms.

------------------------------------------------------------
```

```
I= [rt38,rt37,rt34]
------------------------------------------------------------
Working on rt38 defined as: [[],[38|rt34],[rt28|rt20]]
Calling make_deterministic(Def=[[],[38|rt34],[rt28|rt20]], NewDef)...
Done make_deterministic(Def=[[],[38|rt34],[rt28|rt20]],NewDef=[[],[rt28|
    rt39]])
in 11431.421999999991ms.

Working on rt37 defined as: [[],[38|rt34],[rt24|rt38]]
Calling make_deterministic(Def=[[],[38|rt34],[rt24|rt38]], NewDef)...
Done make_deterministic(Def=[[],[38|rt34],[rt24|rt38]],NewDef=[[],[rt24|
    rt48]])
in 0.0060000000084983185ms.

Working on rt34 defined as: [[],[38|rt34],[rt30|rt35]]
Calling make_deterministic(Def=[[],[38|rt34],[rt30|rt35]], NewDef)...
Done make_deterministic(Def=[[],[38|rt34],[rt30|rt35]],NewDef=[[],[rt144|
    rt118]])
in 0.005000000004656613ms.

------------------------------------------------------------
I= [rt703]
------------------------------------------------------------
Working on rt703 defined as: [[],[rt13|rt32],[rt698|rt703]]
Calling make_deterministic(Def=[[],[rt13|rt32],[rt698|rt703]], NewDef)...
Done make_deterministic(Def=[[],[rt13|rt32],[rt698|rt703]],NewDef=[[],[
    rt698|rt703]])
in 0.15899999999965075ms.

------------------------------------------------------------
I= [rt709,rt708,rt705]
------------------------------------------------------------
Working on rt709 defined as: [[],[38|rt705],[rt426|rt375]]
Calling make_deterministic(Def=[[],[38|rt705],[rt426|rt375]], NewDef)...
Done make_deterministic(Def=[[],[38|rt705],[rt426|rt375]],NewDef=[[],[
    rt426|rt710]])
in 781043.187ms.

Working on rt708 defined as: [[],[38|rt705],[rt389|rt709]]
Calling make_deterministic(Def=[[],[38|rt705],[rt389|rt709]], NewDef)...
Done make_deterministic(Def=[[],[38|rt705],[rt389|rt709]],NewDef=[[],[
    rt389|rt731]])
in 0.008000000030733645ms.

Working on rt705 defined as: [[],[38|rt705],[rt701|rt706]]
Calling make_deterministic(Def=[[],[38|rt705],[rt701|rt706]], NewDef)...
Done make_deterministic(Def=[[],[38|rt705],[rt701|rt706]],NewDef=[[],[
    rt701|rt751]])
in 7.757999999914318ms.

------------------------------------------------------------
```

```
I= [rt758]
------------------------------------------------------------
Working on rt758 defined as: [[],[rt698|rt703],[term|rt758]]
Calling make_deterministic(Def=[[],[rt698|rt703],[term|rt758]], NewDef)...
Done make_deterministic(Def=[[],[rt698|rt703],[term|rt758]],NewDef=[[],[
    term|rt758]])
in 0.07099999999627471ms.

------------------------------------------------------------
I= [rt762,rt760]
------------------------------------------------------------
Working on rt762 defined as: [[],[rt736|rt762],[term|rt760]]
Calling make_deterministic(Def=[[],[rt736|rt762],[term|rt760]], NewDef)...
Done make_deterministic(Def=[[],[rt736|rt762],[term|rt760]],NewDef=[[],[
    term|rt760]])
in 0.24599999992642552ms.

Working on rt760 defined as: [[],[rt701|rt761],[term|rt760]]
Calling make_deterministic(Def=[[],[rt701|rt761],[term|rt760]], NewDef)...
Done make_deterministic(Def=[[],[rt701|rt761],[term|rt760]],NewDef=[[],[
    term|rt760]])
in 0.2900000000372529ms.

{analyzed by plai using eterms with local-control off in 702338.5229999999
    msec.}
}
792662.5410000001 ms.

typeslib: get_type_name/2:
10 calls  0.009000000078231096 ms.    0.0009000000078231096 ms/call
typeslib: new_type_name/1:
6 calls   0.02500000000873115 ms.   0.0041666666681218585 ms/call
typeslib: dz_equivalent_types/2:
12 calls  0.6720000000204891 ms.    0.05600000000170743 ms/call
typeslib: type_escape_term_list/2:
14 calls  0.036999999880208634 ms.    0.002642857134300617 ms/call
typeslib: equivalent_to_top_type/1:
28 calls  0.03600000012374949 ms.   0.0012857142901339103 ms/call
regtype_basic_lattice: set_top_type/1:
39 calls  0.010999999853083864 ms.    0.00028205127828420163 ms/call
typeslib: type_intersection_2/3:
27 calls  0.08599999999569263 ms.   0.003185185185025653 ms/call
typeslib: retract_type_name/3:
33 calls  0.039000000440864824 ms.    0.001181818195177722 ms/call
typeslib: insert_type_name/3:
39 calls  0.05799999993178062 ms.   0.0014871794854302723 ms/call
typeslib: normalize_type/2:
27 calls  0.11100000007718336 ms.   0.004111111113969754 ms/call
regtype_basic_lattice: top_type/1:
32 calls  0.012000000104308128 ms.    0.000375000003259629 ms/call
typeslib: resetunion/0:
```

```
16 calls  0.893999999971129 ms.   0.05587499999819556 ms/call
typeslib: type_union/3:
16 calls  7.977000000042608 ms.   0.498562500002663 ms/call
typeslib: lnewiden_el/4:
16 calls  792485.048 ms.    49530.3155 ms/call
typeslib: get_canonical_name/2:
202 calls   0.13299999920127448 ms.   0.0006584158376300716 ms/call

Total recorded time: 792495.1479999997 ms.

output

{written file /home/daniel/Desktop/IMDEA/ciao-devel/bndls/typeslib/tests/
   lpdoc_problem_eterms_co.pl}
156882.11599999992 ms.
simplify_step2 took 156874.162ms.
```

Listing 4.19: *Records of the slow tests showing calls to make_determ_wide_rules, the types to make deterministic by each call and the time it takes.*

4.3.1 Conclusions

We detected a problem in the configuration of Eterms widening algorithm that could make some bad cases fall into an infinite ascending chain of type approximations: the counter for the number of approximations made for a type name, introduced to guarantee termination, was disabled and we could fix it.

We were able to identify, using the `record_calls` package the problematic predicates for the cost of Eterms analysis. The problem was in the determinization process for types whose definition had been altered during the widening operation. The determinization operation generates, in bad cases, a lot of complex and deep new types. Complex types make basic type operations like type union derivation or type inclusion checks very slow. Additionally, these two operations are key components to the implementation of the determinization predicates.

Due to these findings, we introduced a few optimizations for the tabling system of the type union and inclusion operations, but the performance improvements were not significant enough and therefore left out of the discussion.

Finally, the results of Listing 4.19 (output generated in 157 s. while analysis completed in 793 s.) contrast with the ones shown in Table 4.1. For the fast tests, the type simplification

process done during the output stage was responsible for most of the cost, while in the slow one, most of the cost is due to the analysis.

These findings are what justify the prioritization of optimizing the analysis algorithm widening operation over the type simplification algorithms of Typeslib.

In the following section we discuss possible improvements and solutions to the problems discussed in this Thesis with the objective of achieving the best possible precision/cost trade-off in regular type inference analyses within CiaoPP.

Chapter 5

Proposed Improvements and Solutions

In the previous sections, we have presented the main components of Typeslib, how regular type inference analysis works, the different widening operations included in Typeslib, and which are the most important efficiency liabilities of Eterms.

Taking the current implementation of Eterms as our standard reference for precision and cost of analysis we will now discuss strategies to obtain better trade-offs between precision and efficiency, or to simply obtain better performance.

Sadly, we did not have time to properly implement, measure, and assess the actual impact of the following proposals within the time available for this thesis. Therefore, their implementation and testing will be categorized as future work.

5.1 Improving the Efficiency of Some Type Operations

The most obvious way to achieve better costs while retaining the same precision is to have more efficient type operations. Some improvements over the basic Dart and Zobel algorithms have already been implemented in Typeslib; namely, tabling predicates that store the results of type operations in Ciao's database to avoid redundant computations.

Improving the efficiency of basic type operations will have an impact on the efficiency of all other predicates of Typeslib that rely on them to perform their operations.

5.1.1 Union Operation, Inclusion Test, and Determinization

Most of the cost of Eterms widening lies in the `make_deterministic`, `type_union`, and `dz_type_included` predicates.

`make_deterministic` processes a list of ordered regular type terms (defining a regular type) and tries to combine them functor-wise to achieve a deterministic definition for the type. However, it compares and, when necessary, combines the type terms by pairs, leading to very imbalanced type definition trees (as explained in Section 4.3).

`type_union` first checks if the union has already been computed (tabling), then checks for inclusion of either of the operands in the other; and, as a last resort, creates a new type name and definition for the union by combining the type terms from both original definitions and making them deterministic.

With this premise it should be possible to implement, in a similar fashion, a type union for more than two types at the same time: merging all definitions together and making them deterministic in a single step. This would generate more balanced and shallower type definition trees, which are more efficient in time and memory for computation. Additionally, if we sort the input types by name for this union operation, we could maintain the tables of already computed unions.

For the inclusion checking strategy (needed to avoid redundant types on the database) a tournament-like one could be implemented with linear cost in the worst case:

Regular types are compared in pairs for inclusion checks; if a type includes another, the latter is removed from the union and the first one passes to the next round; if no inclusion relation exists, both types are kept in the union. If an odd number of types need to be paired, one of them automatically advances.

At the end of the tournament, two outcomes are possible: if no type includes the other, the union is a new type and we proceed with its computation; but if a type includes its final opponent it will be our candidate for the global union. In that case, we need to perform inclusion checks for all other types that remain in the union until one of them fails or all

of them are included in our candidate. If an inclusion test fails we compute the union of the remaining types as a new type; but if all other types are included in the candidate, the union is our candidate and we are done.

The original implementations of `make_deterministic` and `type_union` (as well as some of our proposals for improvement) heavily rely on `dz_type_included`. Any improvement effort made on this predicate will have a great positive impact in the rest of the library and in Eterms in particular. Up until now, the proposed improvements for this operation are based on simplifying the types and making them shallower before applying the algorithm.

The more unrolled type definitions are, the easier they are to treat by comparisons such as inclusion. Unrolling a type name within a type definition means substituting references to that type name by the type terms that define it. Of course this has a cost in memory; exploring this trade-off and achieving an optimal unrolling policy could greatly improve the efficiency of these operations.

5.1.2 Inductive Tabling

Another aspect of the implementation of the regular type operations worth reviewing would be the tabling policies. In its current state, whenever a relationship between two or more types is computed, its result is stored in Ciao's database to avoid computing it again in the future. This technique builds tables of computed relations but does not infer extra information.

We believe that adding some inductive steps to the stored data could allow the analyzer to directly skip some costly computations. For example, if $t_1 \subset t_2$ is already asserted in the database, and we compute $t_0 \subset t_1$, we could also assert $t_0 \subset t_2$, $t_0 \cup t_1 = t_1$ and $t_0 \cup t_2 = t_2$ in the database. The cost of asserting these relationships is relatively cheap, but it comes at the risk of it being useless (whenever the asserted operation is not performed in the future). Moreover, the cost of inferring all relationships derived from actually performed computations is not manageable in either memory or processing power. Experiments should

be performed on a broad database (perhaps the entirety of Ciao's code), to figure out what the best inferring policy is.

5.2 Alternative Representations

As we explained in Chapter 2 regular types can be represented in several different forms. Different representations employ their own algorithms for the same operations and differences in efficiency between them are to be expected.

Changing representations for different operations or keeping simultaneous type representations in the database probably would be too costly to obtain any reasonable efficiency improvement. However, efficient libraries for regular type manipulation exist for other programming languages and could be worth exploring.

5.2.1 Libvata

Libvata [15] is an efficient tree automata manipulation library coded in C++ [1]. One approach worth trying would be to implement bridges between Libvata and Typeslib to represent types as tree automata and employ the efficient library operations during analysis. Were the results obtained positive, similar representation and operations could be implemented directly in Typeslib.

5.3 Dynamic Widening Selection

The previous ideas discussed are aimed at maintaining full analysis precision while looking for efficiency improvements. Now we will discuss trade-off options, where some precision is given away to pay for a faster analysis.

The logs from Listing 4.19 show that there is a huge disparity in the time cost of different calls to `make_deterministic` (see lines 34, 42, 50, 76, 86 or 102). Knowing that in most cases Eterms widening is not problematic at all, we could still use it as the default widening

[1] The library can be downloaded from https://github.com/ondrik/libvata

operator for type inference analysis and switch to a simpler, faster, but less precise widening operator for bad cases.

In Section 3.1 we presented the range of widening operations implemented in Typeslib. Any of those could be applied in bad cases for structural type widening and the only remaining question would be how to automatically decide when to swap the widening operator. The simplest idea would be to implement a timer, dependent on the size of the types, and swap if the defined types widening operation does not finish before the timer expires.

A more efficient solution comes with a higher implementation cost; further studies on the detection of bad cases for structural type widening could be performed based on the shape of the types. This could, in the best case, result in a solid prediction system that recommends the best widening operator directly. However, there are no guarantees that such patterns can be used to identify bad cases.

5.4 Bottom-Up Directed Top-Down Widening

The last proposal of this thesis is a new widening operator that aims to keep the simplicity of Deftypes's defined types widening while improving its precision with a bottom-up type inference analysis.

As stated in 3.1.1, the lack of landmarks for type widening operations forces their design to be much more creative and contrived than widenings in most abstract domains. One cannot simply define a finite list of types as the go-to references for all analyses and then widen derived types to their least upper bound amongst them.

The majority of widening operators described in that section perform a sort of local structural search to find references and build generalizations from them; usually resulting in recursive structures.

The main contribution of structural type widening 3.1.2, implemented in Eterms, is that it performs a top-down analysis that uses type names to build global structural references as it traverses the program. These references can then be used to distinguish actual recursion

cases from repetitive structures which are not recursive. This advantage, however, comes with the cost of being too precise and having to check excessively complex type structures at later stages of the analysis.

A much efficient but less precise approach to type inference was presented in the defined types widening 3.1.3. This widening operation uses a list of predefined types as well as the ones explicitly defined in the program as references and widens types to the smallest type reference that contains them. This is a way to circumvent the lack of landmarks for widening operations that we presented at the beginning of this section.

The goal of our **Bottom Up-Directed, Top-Down Widening** is to build more precise and useful landmarks so that an analysis like Deftypes becomes more precise while preserving its good efficiency properties.

To achieve this we could run a bottom-up type inference analysis[2] on the program to analyze, take the resulting data structures as our type landmarks, and apply a top-down analysis like Deftypes approximating derived types to them. This should improve the precision of the top-down analysis while only adding the relatively cheap cost of the bottom-up type inference analysis.

[2]Such as, e.g., Gallaher's analyses, [28] and later papers, which are implemented in CiaoPP 0.8 or accessible at http://akira.ruc.dk/~jpg/Software/bu.html

Chapter 6

Conclusions and Future Work

We provided extensive theoretical background on regular types, their representation, CiaoPP's regular type analyses, and relevant widening operators.

We implemented a property based testing tool to check correctness of basic operations on regular types. We also implemented two regular type generators to create our test cases for the correctness tests. Our test design had the potential limitation of relying on other type operations to assess the correctness of the current operation under test, resulting in some cases that were checking consistency between operations rather than correctness. However, in most cases we only relied on previously tested operations and we were able to obtain valuable information.

Correctness tests of basic operations allowed us to identify and correct a few bugs in the management of the type database. We were able to identify undocumented preconditions for correctness in type operations of the Typeslib library. We could also find examples of Dart an Zobel's inclusion check algorithm not being generally correct, a result presented in [19].

We implemented the `record_call` package for Ciao, which allowed us to easily monitor targeted predicates at run time and log information on their number of calls, average, and total run time. We used this package to monitor the execution of several regular type inference analyses with different CiaoPP domains and confirmed that Eterms generally obtained better precision at a higher cost (as expected).

We tested Eterms type inference analysis on incrementally more complex versions of the same module to obtain information on the escalation of analysis costs. We concluded that almost all of the cost in bad cases was due to the widening operators and the type simplification process after analysis. Even if the cost of analysis was generally better than the simplification cost for the fast cases; in the really bad cases most of the cost fell on the analysis stage. Therefore, we decided to focus our attention on that part of the code.

The `record_call` package allowed us to refine the search by increasing the granularity of our predicate monitorization, leading to the identification of three predicates (`make_deterministic`, `type_union`, and `dz_type_included`) as the responsible ones for the elevated widening costs.

We concluded that the poor performance of these predicates appeared whenever type definitions became more complex, deep, and wide. We also concluded that even in very slow analyses, most of these operations were as fast as usual, and the cost only shot up in some specific instances. This led to the idea of dynamic widening operators, allowing the analyzer to switch to a different widening in these few bad cases, preserving most of the precision.

During this testing process we could also identify and correct a bug in the implementation of Eterms' structural type widening. The bug had to do with the type name approximation counter not working properly and therefore enabling the derivation of infinite ascending chains of approximations. We also introduced some minor improvements to the tabling policies and achieved small performance improvements.

We identified the need to review the simplification policies for the type database, but decided to center our improvement proposals and ideas on the analysis *per se*. Some of the ideas proposed therefore, however, also help keep a simpler database; e.g., switching form pair-wise union/determinization to group-wise operations.

Finally we presented a series of ideas to improve the efficiency of some type operations, like the previously mentioned group-wise union and functor determinization; or the efficient algorithm for deciding whether the union of a set of types belongs to said set. We

also identified the need to study different type unrolling and tabling policies for efficient memory/processing trade-offs in these operations. In this regard, we introduced the idea of implementing inductive tabling policies.

We also concluded that alternative type representations could be useful to achieve better efficiency in some operations and pointed to Libvata's tree automata operations as a possible improvement.

Finally, we proposed two alternatives to the current structural type widening of Eterms. The first idea was to explore policies that allowed the dynamic application of different widening operators to the cases that are bad for Eterms, losing precision only in those cases.

The second one is the novel idea of a bottom-up directed, top-down widening policy. Combining Ciao's defined types widening to widen inferred type approximations to predefined target types with bottom-up type inference analysis to generate the list of target types for the widening.

Unfortunately, the ideas presented for improvement could not be implemented and tested properly during the thesis. This is perhaps the most direct source of future lines of investigation: to take the proposed improvements in Section 5, implement them and test their impact on analysis cost and precision. Among them, we find the implementation of the bottom-up directed defined types widening from Section 5.4 the most motivating and promising. This would not be difficult since these bottom up analyses were already integrated in CiaoPP-0.8 and porting them to the current version of CiaoPP would be quite straightforward (and useful by itself). This has not been done already simply because of lack of time.

While most of the conclusions were clear after the implementation and deployment of our testing tools, some open problems remain: we could not devote time to exploring a characterization of the programs that can perform poorly during analysis. Achieving said characterization would allow an a priori, static implementation of our dynamic widening idea.

The issue of achieving and maintaining an optimal type simplification policy across analyses also remains open and could yield some important performance boosts to the "fast" analysis cases. Similarly, we also have no clear answer to the memory/processing cost trade off derived from considering different degrees on type unrolling in the database. These two lines can be difficult to optimize but could easily find sub-optimal configurations, that still improved performance.

Bibliography

[1] F. Bueno, P. Lopez-Garcia, and M. V. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[2] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: http://tata.gforge.inria.fr/, 2007. release October, 12th 2007.

[3] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Sixth ACM Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979.

[4] P. W. Dart and J. Zobel. A regular type language for logic programs., 1992.

[5] S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[6] S. K. Debray, N.-W. Lin, and M. V. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. 1990 ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 174–188. ACM Press, June 1990.

[7] S. K. Debray, P. Lopez-Garcia, M. V. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[8] Cormac Flanagan. Hybrid Type Checking. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006, pages 245–256. ACM, 2006.

[9] J.P. Gallagher and D.A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming (ICLP'94)*, pages 599–613. MIT Press, 1994.

[10] M. V. Hermenegildo, F. Bueno, M. Carro, E. López-García, P.and Mera, J.F. Morales, and G. Puebla. An overview of ciao and its design philosophy. *Theory and Practice of Logic Programming*, 12(1-2):219–252, 2012.

[11] M. V. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.

[12] M. V. Hermenegildo, G. Puebla, F. Bueno, and P. Lopez-Garcia. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[13] P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.

[14] G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

[15] O. Lengál. An efficient finite tree automata library. *arXiv preprint arXiv:1204.3240*, 2012.

[16] T. Lindgren and P. Mildner. The impact of structure analysis on Prolog compilation. Technical Report 140, Computing Science Departament, Uppsala University, April 1997.

[17] J. W. Lloyd. *Foundations of Logic Programming*. Srpinger-Verlag, second edition, 1987.

[18] P. Lopez-Garcia, F. Bueno, and M. V. Hermenegildo. Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information. *New Generation Computing*, 28(2):117–206, 2010.

[19] L. Lu. On dart-zobel algorithm for testing regular type inclusion. *ACM SIGPLAN Notices*, 36(9):81–85, 2001.

[20] P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study*. PhD thesis, Computing Science Department - Uppsala University, Uppsala, 1999.

[21] P. Mishra. Towards a Theory of Types in Prolog. In *International Symposium on Logic Programming*, pages 289–298, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[22] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *International Conference on Logic Programming (ICLP 1991)*, pages 49–63. MIT Press, June 1991.

[23] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[24] F. Pfenning. *Types in logic programming*. MIT Press Cambridge, Massachusetts, USA, 1992.

[25] G. Puebla, F. Bueno, and M. V. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski,

editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 63–107. Springer-Verlag, September 2000.

[26] G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[27] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin M. Bierman, and Panagiotis Vekris. Safe & Efficient Gradual Typing for TypeScript. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 167–180. ACM, 2015.

[28] H. Saglam and J. Gallagher. Approximating Constraint Logic Programs Using Polymorphic Types and Regular Descriptions. Technical Report CSTR-95-17, Department of Computer Science, University of Bristol, Bristol BS8 1TR, 1995.

[29] A. Serrano, P. Lopez-Garcia, and M. V. Hermenegildo. Resource Usage Analysis of Logic Programs via Abstract Interpretation Using Sized Types. *Theory and Practice of Logic Programming, 30th Int'l. Conference on Logic Programming (ICLP'14) Special Issue*, 14(4-5):739–754, July 2014.

[30] J.G. Siek and W. Taha. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*, pages 81–92, 2006.

[31] Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming*, 29(1–3):17–64, October 1996.

[32] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, pages 102–116. Springer, 2002.

[33] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–153, 1991.